Foundational Nonuniform (Co)datatypes for Higher-Order Logic

Jasmin Christian Blanchette*, Fabian Meier†, Andrei Popescu‡, and Dmitriy Traytel†
*Inria & LORIA, Nancy, France, and Max-Planck-Institut für Informatik, Saarbrücken, Germany
†Institute of Information Security, Department of Computer Science, ETH Zürich, Switzerland
‡School of Science and Technology, Middlesex University London, UK

***Abstract*—Nonuniform (or "nested" or "heterogeneous") datatypes are recursively defined types in which the type arguments vary recursively. They arise in the implementation of finger trees and other efficient functional data structures. We show how to reduce a large class of nonuniform datatypes and codatatypes to uniform types in higher-order logic. We programmed this reduction in the Isabelle/HOL proof assistant, thereby enriching its specification language. Moreover, we derive (co)recusion and (co)induction principles based on a weak variant of parametricity.**

I. Introduction

Inductive (or algebraic) datatypes—often simply called *datatypes*—are a central feature of typed functional programming languages and of most proof assistants. A simple example is the type of finite lists over a type parameter α, specified as follows (in a notation inspired by Standard ML):

$$\alpha\ \mathit{list} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\alpha\ \mathit{list})$$

A datatype is *uniform* if the recursive occurrences of the datatype have the same arguments as the definition itself, as is the case for *list*; otherwise, the datatype is *nonuniform*. Nonuniform types are also called "nested" or "heterogeneous" in the literature. Powerlists are nonuniform:

$$\alpha\ \mathit{plist} = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ \mathit{plist})$$

The type $\alpha\ \mathit{plist}$ is freely generated by the constructors $\mathsf{Nil} : \alpha\ \mathit{plist}$ and $\mathsf{Cons} : \alpha \to (\alpha \times \alpha)\ \mathit{plist} \to \alpha\ \mathit{plist}$. When Cons is applied several times, the product type constructors ($\times$) accumulate to create pairs, pairs of pairs, and so on. Thus, any powerlist of length 3 will have the form

$$\mathsf{Cons}\ a\ (\mathsf{Cons}\ (b_1, b_2)\ (\mathsf{Cons}\ ((c_{11}, c_{12}), (c_{21}, c_{22}))\ \mathsf{Nil}))$$

Nonuniform datatypes arise in the implementation of efficient functional data structures, such as finger trees [24], and they underlie Okasaki's bootstrapping and implicit recursive slowdown optimization techniques [37]. Yet many programming languages and proof assistants lack proper support for such types. For example, even though Standard ML allows nonuniform definitions, a typing restriction disables interesting recursive definitions. As for proof assistants, Agda, Coq, Lean, and Matita allow nonuniform definitions, but they are built into the logic (dependent type theory), with all the risks and limitations that this entails [12, Section 1].

For systems based on higher-order logic such as HOL4, HOL Light, and Isabelle/HOL, no dedicated support exists for nonuniform types, probably because they are widely believed to lie beyond the logic's ML-style polymorphism. Building on the well-understood metatheory of uniform datatypes (Section II), we disprove this folklore belief by showing how to define a large class of nonuniform datatypes by reduction to their uniform counterparts within higher-order logic (Section III).

Our constructions allow variations along several axes. They cater for mutual definitions:

$$\begin{aligned}\alpha\ \mathit{ptree} &= \mathsf{Node}\ \alpha\ (\alpha\ \mathit{pforest})\\ \alpha\ \mathit{pforest} &= \mathsf{Nil} \mid \mathsf{Cons}\ (\alpha\ \mathit{ptree})\ ((\alpha \times \alpha)\ \mathit{pforest})\end{aligned}$$

They allow multiple recursive occurrences, with different type arguments:

$$\alpha\ \mathit{plist}' = \mathsf{Nil} \mid \mathsf{Cons}_1\ \alpha\ (\alpha\ \mathit{plist}') \mid \mathsf{Cons}_2\ \alpha\ ((\alpha \times \alpha)\ \mathit{plist}')$$

They allow multiple type arguments, which may all vary:

$$(\alpha, \beta)\ \mathit{tplist} = \mathsf{Nil}\ \beta \mid \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha, \mathit{unit} + \beta)\ \mathit{tplist})$$

Moreover, they allow the presence of datatypes, codatatypes, and other well-behaved type constructors both around the type arguments and around the recursive type occurrences:

$$\alpha\ \mathit{stree} = \mathsf{Node}\ \alpha\ (((\alpha\ \mathit{fset})\ \mathit{stree})\ \mathit{fset})$$

Here, *fset* is the type constructor associated with finite sets.

Furthermore, the constructions can be extended to coinductive (or coalgebraic) datatypes—*codatatypes*. Codatatypes are generally non-well-founded, allowing infinite values. They are often used to model the datatypes of languages with a nonstrict (lazy) evaluation strategy, such as Haskell, and they can be very convenient for some proving tasks. The codatatype definition

$$\alpha\ \mathit{pstream} \overset{\infty}{=} \mathsf{Cons}\ \alpha\ ((\alpha \times \alpha)\ \mathit{pstream})$$

introduces "powerstreams," with infinite values of the form $\mathsf{Cons}\ a\ (\mathsf{Cons}\ (b_1, b_2)\ (\mathsf{Cons}\ ((c_{11}, c_{12}), (c_{21}, c_{22}))\ \ldots))$.

Unlike dependent type theory, higher-order logic requires all types to be nonempty (inhabited). To introduce a new type, we must both provide a construction in terms of existing types and prove its nonemptiness. For example, a datatype specification analogous to the *pstream* codatatype above should be rejected. In previous work [13], we showed how to decide the nonemptiness problem for uniform types—including mutually recursive specifications and arbitrary mixtures of datatypes and codatatypes—by viewing the definitions as a grammar,

with the defined types as nonterminals. Here, we extend this result to nonuniform types (Section IV). This is achieved by encoding the nonuniformities in a generalized grammar, which can decide nonemptiness of the sets that arise in the construction of the nonuniform types.

Once a datatype has been introduced, users want to define functions that recurse on it and carry out proofs by induction involving these functions—and similarly for codatatypes. A uniform datatype definition generates an induction theorem and a recursor. Nonuniform datatypes pose a challenge, because neither the induction theorem nor the recursor can be expressed in higher-order logic, due to its limited polymorphism. For example, induction for *plist* should look like this:

$$\forall Q.\ Q\ \mathsf{Nil} \wedge \big(\forall x\ xs.\ Q\ xs \Longrightarrow Q\ (\mathsf{Cons}\ x\ xs)\big) \Longrightarrow \forall ys.\ Q\ ys$$

However, this formula is not typable in higher-order logic, because the second and third occurrences of the variable Q in need different types: $(\alpha \times \alpha)\ plist \to bool$ versus $\alpha\ plist \to bool$. Our solution is to replace the theorem by a procedure parameterized by a polymorphic property $\varphi_\alpha : \alpha\ plist \to bool$ (Section V). For *plist*, the procedure transforms a proof goal of the form $\varphi_\alpha\ ys$ into two subgoals $\varphi_\alpha\ \mathsf{Nil}$ and $\forall x\ xs.\ \varphi_{\alpha\times\alpha}\ xs \Longrightarrow \varphi_\alpha\ (\mathsf{Cons}\ x\ xs)$. A weak form of parametricity is needed to recursively transfer properties about φ_α to properties about $\varphi_{\alpha\times\alpha}$. Our approach to (co)recursion is similar (Section VI).

All the constructions and derivations are formalized in the Isabelle/HOL proof assistant and form the basis of high-level commands that let users define nonuniform types and (co)recursive functions on them and reason (co)inductively about them (Section VII). The commands are foundational: Unlike all previous implementations of nonuniform types in proof assistants, they require no new axioms or extensions of the logic. An example involving λ-terms in De Bruijn notation demonstrates the practical potential of our approach.

Our main contributions are the following:

- We designed a reduction of nonuniform datatypes to uniform datatypes within the relatively weak higher-order logic, including recursion and induction.
- We adapted the constructions to support codatatypes as well, exploiting dualities.
- We coded the reduction in a proof assistant based on higher-order logic, yielding a first implementation of nonuniform datatypes without dependent types.

The formal proofs, the source code, and the examples are publicly available [11].

II. Preliminaries

A. Higher-Order Logic

We consider classical higher-order logic (HOL) with Hilbert choice, the axiom of infinity, and rank-1 polymorphism. HOL is based on Church's simple type theory [14]. It is the logic of Gordon's original HOL system [18] and of its many successors and emulators, notably HOL4, HOL Light, and Isabelle/HOL.

Primitive *types* are built from type variables $\alpha, \beta, \ldots$, a type *bool* of Booleans, and an infinite type *ind* using the function type constructor; thus, $(bool \to \alpha) \to ind$ is a type. Primitive *constants* are equality $= : \alpha \to \alpha \to bool$, the Hilbert choice operator, and 0 and Suc for *ind*. Terms are built from constants and variables by means of typed λ-abstraction and application.

A *polymorphic type* is a type T that contains type variables. If T is polymorphic with variables $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, we write $\overline{\alpha}\ T$ instead of T. *Formulas* are closed terms of type *bool*. The logical connectives and quantifiers on formulas are defined using the primitive constants—e.g., True as $(\lambda x : bool.\ x) = (\lambda x : bool.\ x)$. Polymorphic formulas are thought of as universally quantified over their type variables. For example, $\forall x : \alpha.\ x = x$ really means $\forall \alpha.\ \forall x : \alpha.\ x = x$. Nested type quantifications such as $(\forall \alpha.\ \ldots) \Longrightarrow (\forall \alpha.\ \ldots)$ are not expressible. Since HOL was designed to support mathematical reasoning, we will express the concepts in standard mathematical language. Occasionally, when we hit the limitations of HOL, we will indicate so.

The only primitive mechanism for defining new types in HOL is *type definition*: For any existing type $\overline{\alpha}\ T$ and predicate $P : \overline{\alpha}\ T \to bool$ such that $\{x : \overline{\alpha}\ T \mid P\ x\}$ is nonempty, we can introduce a type $\overline{\alpha}\ S$ isomorphic to $\{x : \overline{\alpha}\ T \mid P\ x\}$. Upon meeting the definition $\overline{\alpha}\ S = \{x : \overline{\alpha}\ T \mid P\ x\}$, the system first requires the user to prove $\exists x : \overline{\alpha}\ T.\ P\ x$ and then introduces the type $\overline{\alpha}\ S$, the *projection* $\mathsf{Rep}_S : \overline{\alpha}\ S \to \overline{\alpha}\ T$, and the *embedding* $\mathsf{Abs}_S : \overline{\alpha}\ T \to \overline{\alpha}\ S$ such that $\forall x.\ P\ (\mathsf{Rep}_S\ x)$, $\forall x.\ \mathsf{Abs}_S\ (\mathsf{Rep}_S\ x) = x$, and $\forall x.\ P\ x \Longrightarrow \mathsf{Rep}_S\ (\mathsf{Abs}_S\ x) = x$. The nonemptiness check is necessary because all types in HOL must be nonempty [18], [38].

Thus, unlike dependent type theory, HOL does not have (co)datatypes as primitives. However, datatypes [5], [19], [20], [33] and, more recently, codatatypes [43] are supported via derived specification mechanisms. Users can write fixpoint definitions such in ML-style notation, and the system automatically defines the type using nonrecursive type definitions (ultimately appealing to the infinite type *ind* and the function space); defines the constructors and related operators; and proves characteristic properties, such as injectivity of constructors, induction theorems, and recursor theorems.

B. Bounded Natural Functors

In this paper, we take uniform (co)datatypes for granted, thus assuming the availability of types such as $\alpha\ list$. Often it is useful to think not in terms of polymorphic types, but in terms of type constructors. For example, *list* is a type constructor in one variable, sum $(+)$ and product types $(\times)$ are binary type constructors. Most type constructors are not only operators on types but have a richer structure, that of *bounded natural functors* (*BNF*s) [43].

We write $[n]$ for $\{1, \ldots, n\}$ and $\alpha\ set$ for the powertype of α, consisting of sets of α elements; it is isomorphic to $\alpha \to bool$. An n-ary BNF is a tuple $F = (F, \mathsf{map}_F, (\mathsf{set}^i_F)_{i\in[n]}, \mathsf{bd}_F)$, where

- F is an n-ary type constructor;
- $\mathsf{map}_F : (\alpha_1 \to \beta_1) \to \cdots \to (\alpha_n \to \beta_n) \to \overline{\alpha}\ F \to \overline{\beta}\ F$;
- $\mathsf{set}^i_F : \overline{\alpha}\ F \to \alpha_i\ set$ for $i \in [n]$;
- bd_F is an infinite cardinal number

satisfying the following properties:

- (F, map_F) is an n-ary weak-pullback-preserving functor.

- Each set_F^i is a natural transformation between the functor (F, map_F) and the powerset functor (set, image).
- $\forall i \in [n].\ \forall a \in \mathsf{set}_F^i\ x.\ f_i\ a = g_i\ a \Longrightarrow \mathsf{map}_F\ \overline{f}\ x = \mathsf{map}_F\ \overline{g}\ x$.
- $\forall i \in [n].\ \forall x : (\alpha_1, \alpha_2)\ F.\ |\mathsf{set}_{Fi}\ x| \leq \mathsf{bd}_F$.

For example, *list* is a unary BNF, where map_{list} is the standard map function, set_{list} collects all elements occurring in its argument, and bd_{list} is the cardinality of the natural numbers *nat*.

BNFs are closed under uniform (least and greatest) fixpoint definitions [43] and the nonemptiness problem for BNFs constructed by basic functors, fixpoints and composition is decidable [13]. These crucial properties enable a modular approach to mixing and nesting uniform (co)datatypes and deciding if a high-level specification yields valid, i.e., nonempty, HOL types. In addition, BNFs display predicator and relator structure [41]. The *predicator* $\mathsf{pred}_F : (\alpha_1 \to bool) \to \cdots \to (\alpha_n \to bool) \to \overline{\alpha}\ F \to bool$ and the *relator* $\mathsf{rel}_F : (\alpha_1 \to \beta_1 \to bool) \to \cdots \to (\alpha_n \to \beta_n \to bool) \to \overline{\alpha}\ F \to \overline{\beta}\ F \to bool$, are defined from set_F and map_F as follows:

- $\mathsf{pred}_F\ \overline{P}\ x \Leftrightarrow \forall i \in [n].\ \forall a \in \mathsf{set}_F^i.\ P\ a$;
- $\mathsf{rel}_F\ \overline{R}\ x\ y \Leftrightarrow (\exists z.\ (\forall i \in [n].\ \mathsf{set}_F^i\ z \subseteq \{(a,b) \mid R_i\ a\ b\}) \wedge \mathsf{map}_F\ \overline{\mathsf{fst}}\ z = x \wedge \mathsf{map}_F\ \overline{\mathsf{snd}}\ z = y$, where fst and snd are standard projection functions on the product type $\times$.

For *list*, $\mathsf{pred}_{list}\ P\ xs$ states that P holds for all elements of the list xs, and $\mathsf{rel}_{list}\ R\ xs\ ys$ states that xs and ys have the same length and are element-wise related by R.

Relators and predicators are useful to track parametricity [40], [44]. A polymorphic constant $\mathsf{c} : \overline{\alpha}\ F$ is *parametric* if, for all relations $R_i : \alpha_i \to \beta_i \to bool$ for each $i \in [n]$, we have $\mathsf{rel}_F\ \overline{R}\ \mathsf{c}\ \mathsf{c}$—i.e., every two instances of c are related by the lifting of the relations associated with the component types. Parametricity applies not only to BNFs but also to any combination of BNFs using the function space. For polymorphic functions $\mathsf{f} : \overline{\alpha}\ F \to \overline{\alpha}\ G$ between two BNFs, f is parametric if and only if f is a natural transformation (Appendix A).

III. (Co)datatype Defintions

Before describing the reduction of nonuniform (co)datatypes to uniform (co)datatypes in full generality, we start with a simple example that conveys the main idea. The reduction proceeds by defining a larger uniform datatype and carving out a subset that is isomorphic to the desired nonuniform type. To prove that the constructed type is the intended one, we establish the isomorphism between the defined nonuniform type and the right-hand side of its specification.

A. An Example: Powerlists

Okasaki [37, Section 10.1.1] sketches how to mimic nonuniform datatypes using uniform datatypes. He approximates powerlists by the following definitions:

$\mathsf{datatype}\ \alpha\ sh = \mathsf{Leaf}\ \alpha \mid \mathsf{Node}\ (\alpha\ sh \times \alpha\ sh)$

$\mathsf{datatype}\ \alpha\ raw = \mathsf{Nil}_0 \mid \mathsf{Cons}_0\ (\alpha\ sh)\ (\alpha\ raw)$

The type $\alpha\ raw$ corresponds to lists of binary trees $\alpha\ sh$. It is larger than powerlists in two ways: (1) $\alpha\ sh$ allows nonfull binary trees, which cannot arise in a powerlist; (2) $\alpha\ raw$ imposes no restriction on the depth of the binary trees, whereas a powerlist has elements successively of depth 0, 1, 2, ...

Okasaki considers these mismatches as one of two disadvantages of the above encoding. The other disadvantage is that the encoding requires users to insert Leaf and Node coercions to convert an element such as $\big((a,b),(c,d)\big) : (\alpha \times \alpha) \times (\alpha \times \alpha)$ to $\mathsf{Node}\ (\mathsf{Node}\ (\mathsf{Leaf}\ a, \mathsf{Leaf}\ b), \mathsf{Node}\ (\mathsf{Leaf}\ c, \mathsf{Leaf}\ d)) : \alpha\ sh$.

We overcome the first disadvantage by using a type definition. From the *raw* type, we select exactly those inhabitants that correspond to powerlists. To achieve this, we define two predicates, $\boxed{\mathsf{ok}} : nat \to \alpha\ sh \to bool$ and $\mathsf{ok} : nat \to \alpha\ sh \to bool$, as the least predicates satisfying the following rules:

$$\boxed{\mathsf{ok}}\ 0\ (\mathsf{Leaf}\ x) \qquad \boxed{\mathsf{ok}}\ n\ l \wedge \boxed{\mathsf{ok}}\ n\ r \Longrightarrow \boxed{\mathsf{ok}}\ (n+1)\ (\mathsf{Node}\ (l, r))$$

$$\mathsf{ok}\ n\ \mathsf{Nil}_0 \qquad \boxed{\mathsf{ok}}\ n\ x \wedge \mathsf{ok}\ (n+1)\ xs \Longrightarrow \mathsf{ok}\ n\ (\mathsf{Cons}_0\ x\ xs)$$

The predicate $\boxed{\mathsf{ok}}\ n\ t$ holds if and only if t is a full binary tree of depth n, and $\mathsf{ok}\ n\ xs$ ensures that the first element is a full binary tree of depth n, the second of depth $n+1$, etc. The desired type starts at depth 0: $\alpha\ plist = \{xs : \alpha\ raw \mid \mathsf{ok}\ 0\ xs\}$.

The second disadvantage is addressed by hiding the internal construction of $\alpha\ plist$. We define the powerlist constructors $\mathsf{Nil} : \alpha\ plist$ and $\mathsf{Cons} : \alpha \to (\alpha \times \alpha)\ plist \to \alpha\ plist$ in terms of Nil_0 and Cons_0. These definitions will require some additional machinery on the *raw* type.

B. General Type Construction

We assume that the desired nonuniform datatype has a single constructor. Separate constructors are easy to introduce as syntactic sugar [10, Section 4]. For powerlists, the single constructor definition is $\alpha\ plist = \mathsf{Ctor}_{plist}\ (unit + \alpha \times (\alpha \times \alpha)\ plist)$. It corresponds to finding a least solution (up to isomorphism) to the type fixpoint equation $\alpha\ plist \simeq (\alpha, (\alpha\ F)\ plist)\ G$ with $\alpha\ F = \alpha \times \alpha$ and $(\alpha, \beta)\ G = unit + \alpha \times \beta$.

We generalize this setting in multiple dimensions. First, we support a simultaneous definition of an arbitrary number $\mathbf{i}$ of *mutual* nonuniform datatypes. For example, *ptree* and *pforest* from Section I are given by the system of fixpoint equations

$$\alpha\ ptree \simeq (\alpha, (\alpha\ F_1)\ ptree, (\alpha\ F_2)\ pforest)\ G_1$$
$$\alpha\ pforest \simeq (\alpha, (\alpha\ F_3)\ ptree, (\alpha\ F_4)\ pforest)\ G_2$$

where $(\alpha, \beta, \gamma)\ G_1 = \alpha \times \gamma$, $(\alpha, \beta, \gamma)\ G_2 = unit + \beta \times \gamma$, $\alpha\ F_1 = \alpha\ F_2 = \alpha\ F_3 = \alpha$, and $\alpha\ F_4 = \alpha \times \alpha$. We assume that all G's depend on the same type variables, even though the dependence may be spurious, as in the case of G_1 and β.

Second, a type may occur several times on the right-hand side of a definition. We support an arbitrary number $\mathbf{j}$ of such *occurrences*. This feature is used in the $plist'$ type: $\alpha\ plist' \simeq (\alpha, (\alpha\ F_1)\ plist', (\alpha\ F_2)\ plist')\ G$, where $(\alpha, \beta, \gamma)\ G = unit + \alpha \times \beta + \alpha \times \gamma$, $\alpha\ F_1 = \alpha$, and $\alpha\ F_2 = \alpha \times \alpha$.

Finally, the construction supports an arbitrary number $\mathbf{k}$ of *type parameters*. The parameter changes may differ for different type parameters, such as in the *tplist* example: $(\alpha, \beta)\ tplist \simeq (\alpha, \beta, ((\alpha, \beta)\ F_1, (\alpha, \beta)\ F_2)\ tplist)\ G$, where $(\alpha, \beta, \gamma)\ G = \beta + \alpha \times \gamma$, $(\alpha, \beta)\ F_1 = \alpha \times \alpha$, and $(\alpha, \beta)\ F_2 = unit + \beta$. As before for the G's, all F's may depend on all type parameters of the specified nonuniform type.

In the sequel, the indices i, j, and k range over $[\mathbf{i}]$, $[\mathbf{j}]$, and $[\mathbf{k}]$, respectively. Moreover, we abbreviate indexed sequences

using a horizontal bar; for example, $\overline{\alpha}$ stands for $\alpha_1, \ldots, \alpha_{\mathbf{k}}$, and $\overline{\alpha\, F_1}$ stands for $\alpha\, F_{11}, \ldots, \alpha\, F_{1\mathbf{k}}$. It should be clear from the context which index is omitted.

A *definition* of $\mathbf{i}$ mutual nonuniform datatypes T_i is a system of $\mathbf{i}$ type fixpoint equations

$$\overline{\alpha}\, T_i \simeq \left(\overline{\alpha}, (\overline{\alpha\, F_1})\, T_{\sigma(1)}, \ldots, (\overline{\alpha\, F_{\mathbf{j}}})\, T_{\sigma(\mathbf{j})}\right) G_i \qquad (1)$$

where the G_i's are $(\mathbf{k}+\mathbf{j})$-ary BNFs, the F_{jk}'s are $\mathbf{k}$-ary BNFs, and $\sigma : [\mathbf{j}] \to [\mathbf{i}]$ is a monotone surjective function that expresses which of the $\mathbf{i}$ mutual types belongs to which recursive occurrence. The construction generalizes Okasaki's idea and yields $\mathbf{k}$-ary BNFs T_i that are least solutions (up to isomorphism) to equation (1). A uniform datatype definition [10] is a special case with $\mathbf{j} = \mathbf{i}$, $\sigma(i) = i$, and $\overline{\alpha}\, F_{jk} = \alpha_k$.

We start by defining the *shape* types $\overline{\alpha}\, sh_k$ that overapproximate the recursive changes to the type arguments. There is are $\mathbf{k}$ shape types, corresponding to the $\mathbf{k}$ type arguments, and they are mutually recursive uniform datatypes:

$$\overline{\alpha}\, sh_k = \mathsf{Leaf}_k\, \alpha_k \mid \mathsf{Node}_{1k}\, (\overline{\overline{\alpha}\, sh}\, F_{1k}) \mid \cdots \mid \mathsf{Node}_{\mathbf{j}k}\, (\overline{\overline{\alpha}\, sh}\, F_{\mathbf{j}k})$$

For *plist*, the *sh* type is *sh*. In general, each recursive occurrence may change the type arguments in a different way; this is reflected in the different Node constructors.

Next, we define $\mathbf{i}$ uniform mutually recursive datatypes raw_i that recurse through the G_i's in the same way as the T_i's do, except that they keep the type arguments unchanged. The immediate $\overline{\alpha}$ arguments to G_i are replaced by $\overline{\overline{\alpha}\, sh}$:

$$\overline{\alpha}\, raw_i = \mathsf{Raw}_i\, ((\overline{\overline{\alpha}\, sh}, \overline{\alpha}\, raw_{\sigma(1)}, \ldots, \overline{\alpha}\, raw_{\sigma(\mathbf{j})})\, G_i)$$

For every i, we specify subsets of the types $\overline{\alpha}\, raw_i$ that are isomorphic to the nonuniform types T_i, by defining predicates ok_i that characterize the allowed shapes and their changes in the recursion. As in the powerlist example, the definition of ok_i relies on auxiliary predicates $\boxed{\mathsf{ok}}_k : [\mathbf{j}]\ list \to \overline{\alpha}\, sh_k \to bool$ on the shape types. The type of $\boxed{\mathsf{ok}}_k$ shows an important difference to the *plist* example: The first argument is not just a natural number denoting the depth of a full tree but has more structure. We call it the *shadow* of the shape and let Δ stand for $[\mathbf{j}]\ list$. The additional structure is necessary because different Node_{jk} constructors may occur in a single shape element. These occurrences in the full shape trees are layered: All Node constructors right above the Leaf constructors belong to a fixed occurrence j. The next layer of Nodes may belong to a different fixed occurrence j'. The shadow summarizes the occurrence indices. Consider $\mathsf{Cons}_1\, u\, (\mathsf{Cons}_2\, v\, (\mathsf{Cons}_2\, w\, (\mathsf{Cons}_1\, x\, \mathsf{Nil}))) : plist'$. This order of constructors forces x's type to be $\alpha\, F = \alpha\, F_1\, F_2\, F_2$, with $\alpha\, F_1 = \alpha$ and $\alpha\, F_2 = \alpha \times \alpha$. Consequently, x is embedded into $\alpha\, sh$ as $\mathsf{Node}_2\, (\mathsf{map}_{F_2}\, \mathsf{Node}_2\, (\mathsf{map}_{F_2}\, (\mathsf{map}_{F_2}\, \mathsf{Node}_1)\, (\mathsf{map}_F\, \mathsf{Leaf}\, x)))$, whose shadow is $[2,2,1]$.

Formally, the predicates $\boxed{\mathsf{ok}}_k$ are defined together as the least predicates satisfying the rules

$$\boxed{\mathsf{ok}}_k\ []\ (\mathsf{Leaf}_k\, x)$$
$$\mathsf{pred}_{F_{jk}}\, (\boxed{\mathsf{ok}}_1\, u) \ldots (\boxed{\mathsf{ok}}_{\mathbf{k}}\, u)\, f \Longrightarrow \boxed{\mathsf{ok}}_k\, (j \triangleleft u)\, (\mathsf{Node}_{jk}\, f)$$

where $[]$ and $\triangleleft$ are notations for Nil and Cons. To access the recursive components of *sh*, we rely on the predicators associated with the F's. Predicators are monotone. The $\mathbf{i}$ mutual predicates $\mathsf{ok}_i : \Delta \to \overline{\alpha}\, raw_i \to bool$ are defined similarly:

$$\mathsf{pred}_{G_i}\, (\boxed{\mathsf{ok}}_1\, u) \ldots (\boxed{\mathsf{ok}}_{\mathbf{k}}\, u)\, (\mathsf{ok}_{\sigma(1)}\, (1 \triangleleft u)) \ldots (\mathsf{ok}_{\sigma(\mathbf{j})}\, (\mathbf{j} \triangleleft u))\, g \Longrightarrow \mathsf{ok}_i\, u\, (\mathsf{Raw}_i\, g)$$

We access the $\mathbf{k}$ immediate components of the shape type and the $\mathbf{j}$ recursive components of the raw type through the predicator. We write that an element r of type $\overline{\alpha}\, raw_i$ has shadow u if $\mathsf{ok}_i\, u\, r$ holds.

Finally, the nonuniform type T_i can be defined as the subset of raw_i that satisfies the ok_i predicate for the empty shadow: $\overline{\alpha}\, T_i = \{r : \overline{\alpha}\, raw_i \mid \mathsf{ok}_i\, []\, r\}$. Such a type definition introduces a new type and the embedding–projection pairs $\mathsf{Rep}_i : \overline{\alpha}\, T_i \to \overline{\alpha}\, raw_i$ and $\mathsf{Abs}_i : \overline{\alpha}\, raw_i \to \overline{\alpha}\, T_i$. The emerging nonemptiness problem is discussed in Section IV.

We can prove by induction that ok_i is invariant under the map_{raw_i} function.

Lemma 1: $\mathsf{ok}_i\, u\, (\mathsf{map}_{raw_i}\, \overline{f}\, r) \Leftrightarrow \mathsf{ok}_i\, u\, r$.

This property is sufficient to prove that T_i is a BNF. By virtue of being a BNF, T_i can appear around type arguments and recursive type occurrences in future uniform or nonuniform (co)datatype definitions.

C. Nonuniform Constructors

If the type T_i is the nonuniform type that we intended to construct, it should satisfy equation (1). We prove this isomorphism by defining a constructor $\mathsf{Ctor}_i : (\overline{\alpha}, (\overline{\alpha\, F_1})\, T_{\sigma(1)}, \ldots, (\overline{\alpha\, F_{\mathbf{j}}})\, T_{\sigma(\mathbf{j})})\, G_i \to \overline{\alpha}\, T_i$ and a destructor $\mathsf{dtor}_i : \overline{\alpha}\, T_i \to (\overline{\alpha}, (\overline{\alpha\, F_1})\, T_{\sigma(1)}, \ldots, (\overline{\alpha\, F_{\mathbf{j}}})\, T_{\sigma(\mathbf{j})})\, G_i$ and by showing that they are inverses of each other.

Figure 1 gives diagrammatic definitions of Ctor_i (by composing the functions on the outer arrows) and dtor_i (by composing the functions on the inner arrows). All shape and raw types occurring in the diagram are annotated with their shadows. Abs_i can be applied only to raw elements with shadow $[]$.

The unLeaf_k and unRaw_i functions are inverses of the corresponding constructors satisfying $\mathsf{unLeaf}_k\, (\mathsf{Leaf}_k\, a) = a$ and $\mathsf{unRaw}_i\, (\mathsf{Raw}_i\, r) = r$. Note that unLeaf_k is underspecified and (just as Abs_i) may be applied only to Leaf_k shape elements with shadow $[]$. Moreover, the definition must bridge the gap between the types $\overline{\alpha\, F_j}\, raw_{\sigma(j)}$ of shadow $[]$ and $\overline{\alpha}\, raw_{\sigma(j)}$ of shadow $[j]$ (the rightmost arrows in Figure 1). This must happen recursively (even though the constructor Ctor_i itself is not recursive), by inlining the additional Fs into a new layer of the shape type (right above the Leaf constructors) and therefore requires a generalization that transforms an arbitrary shadow u into $u \triangleright j$ (i.e., the list u with the element j appended to it). For each fixed j, inlining is implemented by means of $\mathbf{i}$ mutual primitively recursive functions $\uparrow_{ji} : (\overline{\alpha\, F_j})\, raw_i \to \overline{\alpha}\, raw_i$, whose definition uses $\mathbf{k}$ mutual primitively recursive functions $\boxed{\uparrow}_{jk} : (\overline{\alpha\, F_j})\, sh_k \to \overline{\alpha}\, sh_k$ on the shape type:

$$\boxed{\uparrow}_{jk}\, (\mathsf{Leaf}_k\, f) = \mathsf{Node}_{jk}\, (\mathsf{map}_{F_{jk}}\, \overline{\mathsf{Leaf}}\, f)$$
$$\boxed{\uparrow}_{jk}\, (\mathsf{Node}_{j'k}\, f) = \mathsf{Node}_{j'k}\, (\mathsf{map}_{F_{j'k}}\, \overline{\boxed{\uparrow}_j}\, f)$$
$$\uparrow_{ji} u\, (\mathsf{Raw}_i\, g) = \mathsf{Raw}_i\, (\mathsf{map}_{G_i}\, \overline{\boxed{\uparrow}_j}\, \uparrow_{j\sigma(1)} \ldots \uparrow_{j\sigma(\mathbf{j})}\, g)$$

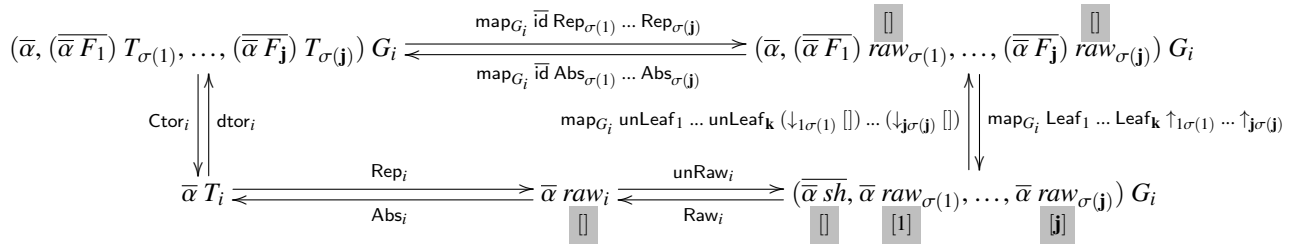

Fig. 1. Definitions of constructors and destructors

Inlining is injective. We define the (partial) inverse operations $\downarrow_{ji} : \Delta \to \overline{\alpha}\ raw_i \to (\overline{\alpha\ F_j})\ raw_i$ and $\boxdot_{jk} : \Delta \to \overline{\alpha}\ sh_k \to (\overline{\alpha\ F_j})\ sh_k$, which are useful when defining the destructors dtor_i. The additional shadow parameter in $\boxdot_{ji}$ denotes how many more layers to destruct until we arrive at the last layer of Nodes (with only Leaf constructors below).

$$\boxdot_{jk}\ []\ (\mathsf{Node}_{jk}\ f) = \mathsf{Leaf}_k\ (\mathsf{map}_{F_{jk}}\ \overline{\mathsf{unLeaf}}\ f)$$
$$\boxdot_{jk}\ (j' \triangleleft u)\ (\mathsf{Node}_{j'k}\ f) = \mathsf{Node}_{j'k}\ (\mathsf{map}_{F_{j'k}}\ \overline{(\boxdot_j\ u)}\ f)$$
$$\downarrow_{ji} u\ (\mathsf{Raw}_i\ g) = \mathsf{Raw}_i\ (\mathsf{map}_{G_i}\ \overline{(\boxdot_j\ u)}(\downarrow_{j\sigma(1)}\ (1 \triangleleft u))\ \ldots\ (\downarrow_{j\sigma(\mathbf{j})}\ (\mathbf{j} \triangleleft u))\ g)$$

We establish the expected behavior of $\uparrow_{ji}$ and $\downarrow_{ji}$ with respect to shadows and prove that they are mutually inverse. The proofs proceed by induction on the *raw* type using very similar omitted auxiliary lemmas for $\boxplus_{jk}$ and $\boxdot_{jk}$.

Lemma 2:

1) $\mathsf{ok}_i\ u\ r \Longrightarrow \mathsf{ok}_i\ (u \triangleright j)\ (\uparrow_{ji} r)$; 3) $\mathsf{ok}_i\ u\ r \Longrightarrow \downarrow_{ji} u\ (\uparrow_{ji} r) = r$;
2) $\mathsf{ok}_i\ (u \triangleright j)\ r \Longrightarrow \mathsf{ok}_i\ u\ (\downarrow_{ji} u\ r)$; 4) $\mathsf{ok}_i\ (u \triangleright j)\ r \Longrightarrow \uparrow_{ji} (\downarrow_{ji} u\ r) = r$.

Since every pair of arrows in Figure 1 is mutually inverse (when applied to elements of the right shadow), we obtain our desired isomorphism property for Ctor_i and dtor_i.

Theorem 3: $\mathsf{dtor}_i\ (\mathsf{Ctor}_i\ g) = g$ and $\mathsf{Ctor}_i\ (\mathsf{dtor}_i\ t) = t$.

Finally, we prove characteristic theorems for T_i's BNF constants. We focus on the property that map_{T_i} commutes with the constructor Ctor_i. The theorems for the relator, the predicator, and the set functions are proved analogously.

Theorem 4: $\mathsf{map}_{T_i}\ \overline{f}\ (\mathsf{Ctor}_i\ g) = \mathsf{Ctor}_i\ (\mathsf{map}_{R_i}\ \overline{f}\ g)$ where $\overline{\alpha}\ R_i = (\overline{\alpha}, (\overline{\alpha\ F_1})\ T_{\sigma(1)}, \ldots, (\overline{\alpha\ F_{\mathbf{j}}})\ T_{\sigma(\mathbf{j})})\ G_i$ and map_{R_i} is the map function associated to this composite BNF.

The proof of Theorem 4 relies on commutation properties of map_{raw_i} and $\uparrow_{ji}$ and of map_{sh_k} and $\boxplus_{jk}$ that can be proved by induction. This is a pervasive pattern when defining recursive functions on nonuniform datatypes.

Lemma 5: $\mathsf{map}_{sh_k}\ \overline{f}\ (\boxplus_{jk}\ s) = \overline{\boxplus_{jk}\ (\mathsf{map}_{sh_k}\ \overline{(\mathsf{map}_{F_j}\ \overline{f})}\ s)}$ and $\mathsf{map}_{raw_i}\ \overline{f}\ (\uparrow_{ji} r) = \uparrow_{ji} (\mathsf{map}_{raw_i}\ \overline{(\mathsf{map}_{F_j}\ \overline{f})}\ r)$.

D. Nonuniform Codatatypes

The construction can be gracefully extended to support types whose elements may be infinitely deep: codatatypes. Given a definition as in equation (1), codatatypes are exactly the types T_i that are the greatest solution to this equation.

This change in semantics needs to be reflected only at the raw level. Accordingly, the raw_i types are defined as an uniform mutual codatatype definition. The shape types remain unchanged, since even in an infinitely deep object all type arguments are finite (but unbounded) type expressions.

The subsequent changes are similarly mild: the predicates ok_i are now defined as a mutual greatest (or coinductive) fixpoint of the same introduction rule; the functions $\uparrow_{ji}$ and $\downarrow_{ji}$ are defined by primitive corecursion, using the same equations as in the datatype case though.

All theorems from Subsections III-B and III-C hold exactly as stated also for codatatypes. The proofs however are different: for example while propositions 1) and 2) of Lemma 2 were proved by induction on r for datatypes, for codatatypes the proof proceeds by coinduction on the now coinductive definitions of ok_i. Similarly, the equational statements (e.g., 3) and 4) of Lemma 2 or the raw part of Lemma 5) are proved by coinduction on the $=$ predicate.

IV. The Nonemptiness Problem

All types in HOL are required to be nonempty. This is a fundamental design decision connected to the presence of Hilbert choice in HOL [18], [38]. As we are developing more sophisticated high-level datatype specification mechanisms, the problem of establishing nonemptiness of the introduced types becomes more difficult.

For nonuniform (co)datatypes T_i specified mutually recursively, the question is whether T_i are indeed valid HOL types, i.e., are nonempty. We are interested in an answer that is both *automatic*, i.e., is given without asking the user to perform any proof, and *complete*, i.e., does not reject any valid types.

In previous work, we designed a solution for mutual uniform (co)datatypes [13]. It is based on storing, for each BNF $\overline{\alpha}K$ with $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$, complete information on its *conditional nonemptiness*, i.e., on which combinations of nonemptiness assumptions for the argument types α_i would be sufficient to guarantee nonemptiness of $\overline{\alpha}K$. For example, if $n = 3$ and $\overline{\alpha}K$ is $\alpha_1\ stream + \alpha_2 \times \alpha_3$, then for $\overline{\alpha}K$ to be nonempty it suffices that either α_1, or both α_2 and α_3 be nonempty. We say that both $\{\alpha_1\}$ and $\{\alpha_2, \alpha_3\}$, or, simply, $\{1\}$ and $\{2,3\}$ are *witnesses* for the nonemptiness of $\alpha_1\ stream + \alpha_2 \times \alpha_3$.

The above discussion assumes that K operates on possibly nonempty collections of elements (which, technically, as a type constructor, it does not, since the HOL type variables are assumed to range over nonempty types). To model this, we employ the set_K operators to capture the action of K on *sets*, as the homonymous constant $K : \alpha_1\ set \to \ldots \to \alpha_n\ set \to (\overline{\alpha}K)\ set$, defined by $K\ A_1\ \ldots\ A_n = \{x : \overline{\alpha}K \mid \forall i \in [n].\ \mathsf{set}^i_K\ x \subseteq A_i\}$. This way, the constant K operates on sets like the type

constructor K operates on types. And since sets can be empty, we are able to express witnesshood:

Given $I \subseteq [n]$, we call I a *witness for* K if, for all sets $\overline{A}$, $\forall i \in I.\, A_i \neq \varnothing$ implies $K\,\overline{A} \neq \varnothing$. A set $\mathscr{I} \subseteq [n]\,set$ of witnesses for K is called *perfect* if for all witnesses $J \subseteq [n]$ there exists $I \in \mathscr{I}$ such that $I \subseteq J$. Thus, a perfect set of witnesses for K is one where no witness is missed, in that any witness J is equal to, or improved by, an $I \in \mathscr{I}$.

We fix a definition of $\mathbf{i}$ mutual nonuniform datatypes T_i, as depicted in equation (1) from Section III-B. We assume the involved BNF's, namely, each G_i and each F_{jk}, are endowed with perfect sets of witnesses $\mathscr{I}(G_i)$ and $\mathscr{I}(F_{jk})$. We will show how to effectively construct perfect sets of witnesses for the T_i's. On the one hand, this allows us to decide when the T_i's are nonempty, hence valid HOL types—if and only if their perfect sets are nonempty. On the other hand, this equips the T_i's with infrastructure needed to establish nonemptiness in future (co)datatypes that may use them as parameters.

To identify the witnesses for the T_i's, we try a similar approach to what we did for uniform datatypes. There, we define a context-free set-grammar (which is like a standard context-free grammar except that its productions act on *finite sets* rather than words) having the T_i's as nonterminals and the argument types α_i as terminals. The productions of the set-grammar followed the direction of the destructors,

$$\overline{\alpha}\; T_i \xrightarrow{\mathsf{dtor}_i} (\overline{\alpha}, \overline{\alpha}T_1, \ldots, \overline{\alpha}T_{\mathbf{i}})\; G_i$$

with each T_i deriving sets containing the nonterminals $T_{i'}$ and the terminals α_k allowed by witnesses of G_i.

For the nonuniform case, when applying recursively productions following the definitions

$$\overline{\alpha}\; T_i \xrightarrow{\mathsf{dtor}_i} (\overline{\alpha}, (\overline{\overline{\alpha}\; F_1})\; T_{\sigma(1)}, \ldots, (\overline{\overline{\alpha}\; F_{\mathbf{j}}})\; T_{\sigma(\mathbf{j})})\; G_i$$

we see that the T_i's become applied to larger and larger polynomial expressions involving the F_{jk}'s. To keep the set-grammar finite, we take a more abstract view, retaining from the F_{jk}-expressions only their witness-relevant information, obtained by suitably combining their perfect sets $\mathscr{I}(F_{jk})$. We define the set $\mathsf{PolyWit}$, of *polynomial witness sets* (*polywits* for short), inductively as follows:

- If $k \in [\mathbf{k}]$, then $\{\{k\}\} \in \mathsf{PolyWit}$.
- If $(j,k) \in [\mathbf{j}] \times [\mathbf{k}]$ and $p_1, \ldots, p_{\mathbf{k}} \in \mathsf{PolyWit}$, then $(p_1, \ldots, p_{\mathbf{k}}) \cdot \mathscr{I}(F_{jk}) \in \mathsf{PolyWit}$.

Note that polywits are sets of subsets of $[\mathbf{k}]$. In the second clause above, we used the composition $(p_1, \ldots, p_{\mathbf{k}}) \cdot \mathscr{I}(F_{jk})$, which is defined as $\bigcup_{I \in \mathscr{I}(F_{jk})} \{\bigcup_{k' \in I} J_{k'} \mid \overline{J} \in \prod_{k' \in I} p_{k'}\}$. This composition captures the computation of witnesses for composed BNFs, namely, for the composition of F_{jk} with the BNFs corresponding to the polywits $p_1, \ldots, p_{\mathbf{k}}$.

We fix a set of tokens, $\mathsf{Tok} = \{t_i \mid i \in [\mathbf{i}]\}$, to represent symbolically the T_i's. We define the set-grammar $\mathsf{Gr} = (\mathsf{Term}, \mathsf{NTerm}, \mathsf{Prod})$ as follows. Its terminals Term are $[\mathbf{k}]$, i.e., one number $k \in [\mathbf{k}]$ for each type variable α_k. The nonterminals NTerm are either polywits or have the form $(p_1, \ldots, p_{\mathbf{k}})\, t_i$, where each p_k is a polywit and $i \in [\mathbf{i}]$. There are two types of productions:

1) $p \Longrightarrow I$, where $p \in \mathsf{PolyWit}$ and $I \in p$
2) $\overline{p}\, t_i \Longrightarrow \Gamma_J$ where $i \in [\mathbf{i}]$, $J \in \mathscr{I}(G_i)$ and $\Gamma_J = \{p_k \mid k \in J \cap [\mathbf{k}]\} \cup \{(\overline{p} \cdot \mathscr{I}(F_{j1}), \ldots, \overline{p} \cdot \mathscr{I}(F_{j\mathbf{k}}))\, t_{\sigma(j)} \mid \mathbf{k} + j \in J\}$

The first type of production selects witnesses from polywits. The second type mirrors the recursion in the definition of the T_i's by following the destructors and selecting the terminals and nonterminals according to the witnesses of G_i.

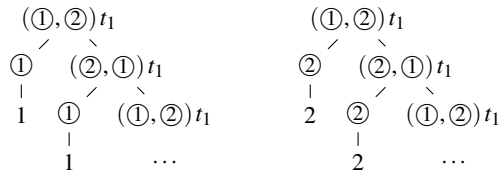

Fig. 2. Derivation trees in the witness grammar

Let $\mathsf{Lang}_i(\mathsf{Gr})$ be the language, i.e., set of subsets of $[\mathbf{k}]$, generated by Gr starting from the nonterminal $(\{\{a_1\}\}, \ldots, \{\{a_{\mathbf{k}}\}\})\, t_i$ (the token for T_i applied to the trivial polywits for its argument types). Similarly, let $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$ be the language cogenerated by Gr—allowing infinite chains of productions, i.e., allowing infinite derivation trees—again, starting from $(\{\{a_1\}\}, \ldots, \{\{a_{\mathbf{k}}\}\})\, t_i$.

Theorem 6:

1) If we interpret the definition as specifying mutual datatypes, then:

1.1) the definition is valid in HOL (i.e., the specified types are nonempty) if and only if $\mathsf{Lang}_i(\mathsf{Gr}) \neq \varnothing$;

1.2) $\mathsf{Lang}_i(\mathsf{Gr})$ is a perfect set of witnesses for T_i.

2) If we interpret the definition as specifying mutual codatatypes, then:

1.1) the definition is always valid in HOL (and in fact $\mathsf{Lang}_{\infty,i}(\mathsf{Gr}) \neq \varnothing$ always holds);

1.2) $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$ is a perfect set of witnesses for T_i.

Note how the formulation of the theorem distinguishes the nonemptiness subproblem from the witness problem. This is because the T_i's cannot be registered as types without knowing their nonemptiness, more precisely, without knowing the nonemptiness of their representing predicates $\mathsf{ok}_i\,[]$ from the raw types raw_i. In the codatatype case, nonemptiness always holds thanks to the greatest fixpoint nature of the construction.

Since the raw_i's are BNFs, we already have a perfect set of witnesses for them, but those usually will not satisfy $\mathsf{ok}_i\,[]$ (and even if they were, they may not give a perfect set for T_i). So to prove the theorem we adapt the notion of witness from types to predicates and show that the languages (co)generated by Gr offer perfect sets for $\mathsf{ok}_i\,u$ for any shadow u. We generalize from $[]$ to arbitrary u because the shadow increases along applications of the raw_i destructors. Appendix B gives details.

For any finite set-grammar Gr, the languages $\mathsf{Lang}_i(\mathsf{Gr})$ and $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$ are effectively computable by fixpoint iteration [13]. Moreover, in [13, Section 4.3] we show that iteration only

needs a number of steps equal to the number of nonterminals. However, for uniform datatypes this number is precisely that of mutual types, **i**; in our nonuniform case, it is the larger number $\mathbf{j} \times \mathbf{k} \times |\mathsf{PolyWit}|$, where $|\mathsf{PolyWit}| = O(2^{2^{\mathbf{k}}})$. Fortunately, in practice the declared types do not have many type variables, so the doubly exponential blowup in **k** is not problematic.

As an example, consider the following contrived definition of the nonuniform codatatype of (α_1, α_2)-alternating streams:

$$(\alpha_1, \alpha_2)\, \mathit{alter} \overset{\infty}{=} \mathsf{C}\, \alpha_1\, ((\alpha_2, \alpha_1)\, \mathit{alter}) \mid \mathsf{D}\, \alpha_2\, ((\alpha_2, \alpha_1)\, \mathit{alter})$$

Thus, we have $\mathbf{i} = 1$, $\mathbf{j} = 1$, $\mathbf{k} = 2$, σ is the unique function from [1] to [1], $(\alpha_1,\alpha_2)F_{11} = \alpha_2$, $(\alpha_1,\alpha_2)F_{12} = \alpha_1$ and $(\alpha_1,\alpha_2,\alpha_3)\,G_1 = \alpha_1 \times \alpha_3 + \alpha_2 \times \alpha_3$. Thus, $\{\{2\}\}$, $\{\{1\}\}$ and $\{\{1,3\},\{2,3\}\}$ are perfect sets of witnesses for F_{11}, F_{21} and G_1, respectively. Figure 2 shows two infinite derivation trees from the initial nonterminal $(①,②)\,t_1$ in the grammar Gr associated to this definition, where we write ① and ② for the polywits $\{\{1\}\}$ and $\{\{2\}\}$. The trees repeat the same pattern after reaching $(①,②)\,t_1$. In the left tree, the top production is $(①,②)\,t_1 \Longrightarrow \{①,(②,①)\,t_1\}$; this is a valid production of type 2, based on G_1's witness $\{1,3\}$. By contrast, the tree's other production of type 2, $(②,①)\,t_1 \Longrightarrow \{①,(①,②)\,t_1\}$, uses the other witness, $J = \{2,3\}$. The frontiers of the two trees are $\{1\}$ and $\{2\}$, respectively. In fact, $\{\{1\},\{2\}\}$ forms a perfect set of witnesses for *alter*.

Even though *alter* is always nonempty, since it is a codatatype, determining a perfect set of witnesses is important for maintaining a complete solution to the overall nonemptiness problem. If we used an imperfect set of witnesses such as $\{\{1,2\}\}$, we would reject valid datatypes such as $\alpha\, \mathit{fractal} = (\alpha, ((\alpha,\alpha)\ \mathit{alter})\ \mathit{fractal})\ \mathit{alter}$, where we must know that $\{\{1\}\}$ is a witness for *alter* to infer nonemptiness.

V. (Co)induction Principles

In a proof assistant, high-level abstractions must be matched by a reasoning apparatus. Next we discuss how reasoning principles for nonuniform (co)datatypes can be inferred in HOL. To avoid cluttering the ideas with too many technicalities, in this and the next section we discuss the restricted situation of a single (co)datatype $\alpha\, T$ defined as fixpoint isomorphism $\alpha\, T \simeq (\alpha, \alpha\, F\, T)\, G$ as in Section III-B but with $\mathbf{i} = \mathbf{j} = \mathbf{k} = 1$. We will reuse all the infrastructure defined in Section III-B while omitting all indices except for when they are needed, e.g., for distinguishing between the two set operators for G.

(Co)induction involves reasoning about the elements of a (co)datatype and those of their (co)recursive *components*. BNFs allow us to capture components abstractly, in terms of the "set" operators. For example, for any element r of the uniform (co)datatype $\alpha\ \mathit{raw}$, its components are the elements r' of $\mathsf{set}^2_G(\mathsf{unRaw}\ r)$—because, in its fixpoint definition, *raw* appears recursively as the second argument of G.

A. Induction

Induction for *uniform* datatypes can be smoothly expressed in HOL. For example, the induction principle for $\alpha\ \mathit{raw}$ is the following HOL theorem, which we will refer to as $\mathsf{Ind}_{\mathit{raw}}$:

$$\forall Q.\ (\forall r : \alpha\ \mathit{raw}.\ (\forall r' \in \mathsf{set}^2_G\ (\mathsf{unRaw}\ r).\ Q\ r') \Longrightarrow Q\ r) \Longrightarrow \forall r : \alpha\ \mathit{raw}.\ Q\ r$$

It states that, to show that a predicate Q holds for all $\alpha\ \mathit{raw}$, it suffices to show that Q holds for any element r given that Q holds for the recursive components $\mathsf{set}^2_G\ (\mathsf{unRaw}\ r)$ of r.

As we remarked in Section I, a verbatim translation of $\mathsf{Ind}_{\mathit{raw}}$ for T would not be typable, since Q would be a variable used with two different types. But even if we change Q from a quantified variable to a polymorphic predicate $\mathsf{Q} : \alpha\ \mathit{raw} \to \mathit{bool}$ (and remove the outer $\forall$), the formula would be unsound, due to the cross-type nature of the T-components: Whereas t has type $\alpha\, T$, its components t' have type $\alpha\, F\, T$. For example, if Q is vacuously false on the type $\mathit{nat}\ F\ T$, we could use such an induction theorem (with α instantiated to *nat*) to wrongly infer that Q is true on $\mathit{nat}\ T$.

On the other hand, for each polymorphic predicate $\mathsf{P} : \alpha\, T \to \mathit{bool}$, we can hope to prove the following *inference rule* in HOL, where for clarity we make explicit the universal quantification over the type variable α, occurring both in the assumption and the conclusion

$$\frac{\forall \alpha.\ \forall t : \alpha\, T.\ (\forall t' \in \mathsf{set}^2_G\ (\mathsf{dtor}\ t).\ \mathsf{P}\ t') \Longrightarrow \mathsf{P}\ t}{\forall \alpha.\ \forall t : \alpha\, T.\ \mathsf{P}\ t}\ \mathsf{Ind}^{\mathsf{P}}_T$$

Let us try to prove this rule sound. All we have at our disposal is the representation type $\alpha\ \mathit{raw}$ and its induction principle. So we should try to reduce $\mathsf{Ind}^{\mathsf{P}}_T$ to $\mathsf{Ind}_{\mathit{raw}}$ along the embedding–projection pair $\mathsf{Rep} : \alpha\, T \to \alpha\ \mathit{raw}$ and $\mathsf{Abs} : \alpha\ \mathit{raw} \to \alpha\, T$, where the predicate $\mathsf{ok}\ []$ describes the image of Rep.

We start by defining Q to be $\mathsf{Abs} \circ \mathsf{P}$ and try to prove $\forall r.\ \mathsf{ok}\ []\ r \Longrightarrow \mathsf{Q}\ r$ using $\mathsf{Ind}_{\mathit{raw}}$, hoping to be able to connect the hypothesis of $\mathsf{Ind}^{\mathsf{P}}_T$ with that of $\mathsf{Ind}_{\mathit{raw}}$. We quickly encounter the following problem, depicted in Figure 3.[1] Suppose $r : \alpha\ \mathit{raw}$

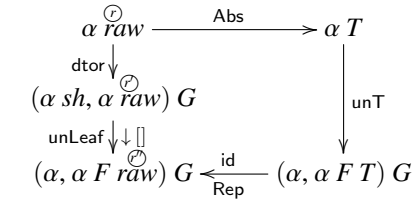

Fig. 3. The *raw* representation of T

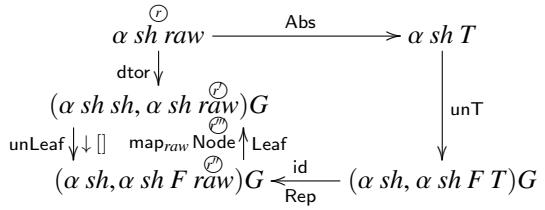

Fig. 4. Connecting the T- and the *raw*- components

[1]There and in forthcoming figures we replace all $\mathsf{map}_G\ f\ g$ arrow annotations with arrows carrying two labels The types should make clear which function represents which argument of map_G. For uniformity, we put the first argument f to the right of the arrow's direction and the second argument g to the left.

corresponds to $t : \alpha\ T$ (via the embedding–projection pair); then T-induction speaks about the T-components $t' : \alpha\ F\ T$ of t, which do *not* correspond to the *raw*-components $r' : \alpha\ raw$ of r, but rather to elements $r'' : \alpha\ F\ raw$ of the form $\downarrow []\ r'$. This mismatch is a consequence of our representation technique: To represent T's destructor using *raw*'s destructor we needed to apply the "correction" $\downarrow []$ for the nonuniformity. In order to cope with it, we appeal to the shape type $\alpha\ sh$, which is in the simplified setting essentially $\alpha + \alpha\ F + \alpha\ F^2 + \ldots$, and thus includes all the types inhabited by t, its components, the components' components and so on.

So we weaken our goal, trying to prove that P holds not on all types $\alpha\ T$, but only on types of the form $\alpha\ sh\ T$—for Q, this means switching from $\alpha\ raw$ to $\alpha\ sh\ raw$. As shown in Figure 4, now we have a way to travel from the type $\alpha\ sh\ F\ raw$ back to the type $\alpha\ sh\ raw$—namely, by applying Node to level the nonuniformity F into the larger type sh. For this to work, we need Q to reflect $\mathsf{map}_{raw}\ \mathsf{Node}$, i.e., have $\mathsf{Q}\ (\mathsf{map}_{raw}\ \mathsf{Node}\ r'')$ imply $\mathsf{Q}\ r''$.

Another issue is that $r''' = \mathsf{map}_{raw}\ \mathsf{Node}\ r''$ itself is not in the image of Rep: r''' has shadow $[1]$ instead of the required $[]$. We must generalize our goal to arbitrary shadows, i.e., to $\forall r\ u.\ \mathsf{ok}\ u\ r \Longrightarrow \mathsf{Q}'\ u\ r$, for a suitable predicate $\mathsf{Q}' : \Delta \to \alpha\ sh\ raw \to bool$ that extends Q in that $\mathsf{Q}'\ [] = \mathsf{Q}$. To this end, we define $\Downarrow : \Delta \to \alpha\ sh\ raw \to \alpha\ sh\ raw$, an operator that generalizes the trip from r' to r'' to r''' described above to an arbitrary shadow u, and $\Downarrow\!\!\Downarrow$, the cumulative iteration of $\Downarrow$:

$$\Downarrow u\ r = \mathsf{map}_{raw}\ \mathsf{Node}\ (\downarrow u\ r) \qquad \begin{array}{ll} \Downarrow\!\!\Downarrow []\ r & = s \\ \Downarrow\!\!\Downarrow (u \triangleright 1)\ r & = \Downarrow u\ (\Downarrow\!\!\Downarrow u\ r) \end{array}$$

To see the intuition of these operators, we can regard the elements of both $\beta\ sh$ and $\beta\ raw$ as trees whose leaves are elements of β and whose nodes branch according to F. Then $\Downarrow$ traverses elements of $\alpha\ sh\ raw$ until it reaches their innermost nodes (with only leaves, i.e., elements of $\alpha\ sh$, as subtrees), and then immerses them as top nodes in the inner shape layer. The additional shadow argument u is needed in oder to identify when an innermost tree has been reached (since we only count on $\Downarrow u\ r$ being well-behaved if $\mathsf{ok}\ u\ r$ holds).

The *sh* counterparts of the above, $\boxed{\Downarrow} : \Delta \to \alpha\ sh\ sh \to \alpha\ sh\ sh$ and $\boxed{\Downarrow\!\!\Downarrow}$, are defined similarly (using map_{sh} instead of map_{raw}). The key property of the "immerse" family of operators is that they commute with *raw*'s destructor in the following sense.

Lemma 7: The left subdiagram in Figure 5 is commutative.

Now, taking $\mathsf{Q}'\ u\ r$ to be $\mathsf{Q}\ (\Downarrow\!\!\Downarrow u\ r)$ does the job. Namely, Q' can be proved by *raw*-induction on r, since it achieves the desired correspondence between the *raw*-components and the T-components, namely, between the leftmost and rightmost edges of Figure 5's diagram. That the correspondence works is ensured by the diagram's commutativity, as a composition of two commutative subdiagrams: the left by the above lemma and the right by the definition of dtor.

Thus, assuming the hypothesis of $\mathsf{Ind}_T^{\mathsf{P}}$, we have proved $\forall\alpha.\ \forall r : \alpha\ sh\ raw.\ \forall u : \Delta.\ \mathsf{ok}\ u\ r \Longrightarrow \mathsf{Q}'\ u\ r$—in particular, $\forall\,\alpha.\ \forall r : \alpha\ sh\ raw.\ \mathsf{ok}\ []\ r \Longrightarrow \mathsf{Q}\ r$, which implies $\forall\alpha.\ \forall t : \alpha\ sh\ T.\ \mathsf{P}\ t$.

However, we had started out to prove the more general fact $\forall\alpha.\ \forall t : \alpha\ T.\ \mathsf{P}\ t$. To move from the former to the latter, it would suffice that P reflects the operator $\mathsf{map}_T\ \mathsf{Leaf} : \alpha\ T \to \alpha\ sh\ T$ in the same way we needed it to reflect the operator $\mathsf{map}_{raw}\ \mathsf{Node}$ earlier: $\forall t''.\ \mathsf{P}\ (\mathsf{map}_T\ \mathsf{Node}\ t'') \to \mathsf{P}\ t''$. In fact, it suffices to assume that P is *injective-antitone-parametric* (*IAP*), in that $\mathsf{P}\ (\mathsf{map}_T\ f\ t)$ implies $\mathsf{P}\ t$ for all $t : \alpha\ T$ and all injective functions $f : \alpha \to \beta$. (Both Leaf and Node are injective.) In conclusion, we have obtained:

Theorem 8: If P is IAP, then $\mathsf{Ind}_T^{\mathsf{P}}$ is derivable in HOL.

IAP is related but significantly weaker than (arbitrary) parametricity, which for P would mean $\mathsf{P}\ t \Leftrightarrow \mathsf{P}\ (\mathsf{map}_T\ f\ t)$ for all t and arbitrary functions f (Appendix A).

Due to the limitations of HOL, we were only able to prove a restricted form of induction. However, all formulas built from the usual terms used in functional programming and employing equality, the logical connectives and universal quantifiers are IAP, and therefore fall in the scope of our theorem. Indeed, all these operators are either fully parametric or IAP—the equality and universal quantification are IAP, but not fully parametric. The main losses are constants defined using Hilbert choice, existential quantifiers, and ad hoc overloaded constants. For example, a predicate $\mathsf{P} : \alpha\ T \to bool$ that has different definitions on $nat\ T$ and $int\ T$ is not IAP.

B. Coinduction

We have designed the above infrastructure, consisting of the "immerse" operators, to work equally well for the codatatype as it does for the datatype. Namely, when $\alpha\ T$ is a codatatype, these operators are defined in the same way and can be used to derive the soundness of a nonuniform coinduction rule under similar assumptions to the induction case (from the corresponding uniform coinduction on the *raw* type):

$$\frac{\begin{array}{l}\forall\alpha.\ \forall t_1, t_2 : \alpha T. \\ \quad \mathsf{P}\ t_1\ t_2 \Longrightarrow \mathsf{rel}_G\ (=)\ \mathsf{P}\ (\mathsf{dtor}\ t_1)\ (\mathsf{dtor}\ t_2)\end{array}}{\forall\alpha.\ \forall t : \alpha T.\ \mathsf{P}\ t_1\ t_2 \Longrightarrow t_1 = t_2}\ \mathsf{Coind}_T^{\mathsf{P}}$$

For this rule to be sound, $\mathsf{P} : \alpha\ T \to \alpha\ T \to bool$ should again interact well with injective functions, however this time in the opposite direction. We say that P is *injective-monotone-parametric* (*IMP*), if $\mathsf{P}\ t_1\ t_2$ implies $\mathsf{P}\ (\mathsf{map}_T\ f\ t_1)\ (\mathsf{map}_T\ g\ t_2)$ for all $t_1, t_2 : \alpha\ T$ and injective functions $f, g : \alpha \to \beta$.

Theorem 9: If P is IMP, then $\mathsf{Coind}_T^{\mathsf{P}}$ is derivable in HOL.

Unlike IAP, IMP disallows the usage of the universal quantifier in P, while it allows the existential quantifier. This is a quite desirable symmetry: Induction requires the universal quantifier to perform generalization over non-inductive parameters. For coinduction, the existential quantifier takes this role.

VI. (Co)recursion Principles

For nonuniform (co)datatypes to be practically useful, there must exist some infrastructure supporting (co)recursive function definitions. We start with datatypes and consider the following simple recursive function on powerlists:

$$\begin{array}{l} \mathsf{split} : (\alpha \times \beta)\ plist \to \alpha\ plist\ \times \beta\ plist \\ \mathsf{split}\ \mathsf{Nil} \qquad\qquad = (\mathsf{Nil}, \mathsf{Nil}) \\ \mathsf{split}\ (\mathsf{Cons}\ ab\ xs) = \mathsf{let}\ (as, bs) = \mathsf{split}\ (\mathsf{map}_{plist}\ \mathsf{swap}\ xs) \\ \qquad\qquad\qquad\qquad\quad \mathsf{in}\ (\mathsf{Cons}\ (\mathsf{fst}\ ab)\ as, \mathsf{Cons}\ (\mathsf{snd}\ ab)\ bs) \end{array}$$

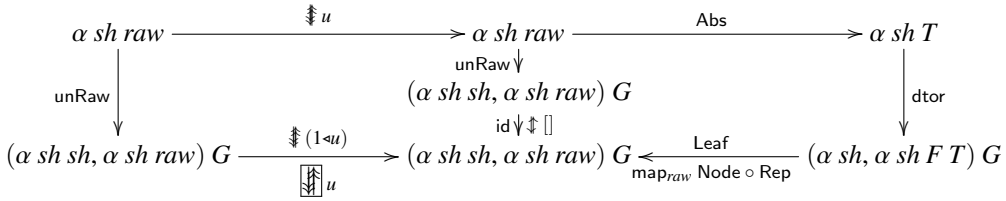

Fig. 5. Borrowing induction and coinduction from *raw* to T

Here, the pattern matched variable *xs* has type $((\alpha \times \beta) \times (\alpha \times \beta))$ *plist* and the auxiliary swap function is defined as $\mathsf{swap}\,((a_1, b_1), (a_2, b_2)) = ((a_1, a_2), (b_1, b_2))$. The function split uses polymorphic recursion: its type on the right hand side of the specification is different from the one on the left hand side. More precisely, the recursive call is applied to an argument of type $((\alpha \times \alpha) \times (\beta \times \beta))$ *plist*. None of the existing tools for defining recursive functions in higher-order logic can handle polymorphic recursion—the gap we are about to fill.

Also note that split is not primitively recursive in the standard sense: the recursive call is applied to a modified pattern matched argument $\mathsf{map}_{plist}\,\mathsf{swap}\,xs$. However, the modification happens through the map_{plist} function, which does not change the length of *xs*. Hence, such *generalized primitively recursive* specifications, which modify the arguments of the recursive call merely through a map function, are safely terminating.

A. Generalized Primitive Recursion

Following the foundational approach, primitively recursive specifications in HOL are reduced to nonrecursive definitions using a recursion combinator [10]. The equally expressive but slightly less convenient *primitively iterative specifications* can be reduced too, using a simpler fold combinator. For a uniform datatype $\alpha\ T = \mathsf{Ctor}\ ((\alpha, \alpha\ T)\ G)$ (e.g., $\alpha\ T = \alpha\ list$ with $(\alpha, \beta)\ G = unit + \alpha \times \beta$) the fold combinator has type

$$((\alpha, \beta)\ G \to \beta) \to \alpha\ T \to \beta$$

A function $\mathsf{f} = \mathsf{fold}\ \mathsf{b}$ for some fixed $\mathsf{b} : (\alpha, \beta)\ G \to \beta$, satisfies the characteristic recursive equation $\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ \mathsf{f}\ g)$. We call b the *blueprint* of f. Note that b describes how to combine the *results* of the recursive calls into a new result of type β. The recursion combinator's blueprint, of type $(\alpha, \alpha\ T \times \beta)\ G \to \beta$, generalizes fold's blueprint by providing access to the original $\alpha\ T$ values, in addition to the results of the recursive calls. Although our ideas support recursion, we focus on iteration to simplify the presentation.

For a nonuniform datatype $\alpha\ T = (\alpha, \alpha\ F\ T)\ G$ (as before for simplicity we consider the setting $\mathbf{i} = \mathbf{j} = \mathbf{k} = 1$), the natural generalization of fold would be a combinator of type

$$\forall Y.\, (\forall \alpha.\, (\alpha, \alpha\ F\ Y)\ G \to \alpha\ Y) \to \beta\ T \to \beta\ Y$$

where the universally quantified type constructor Y captures the positions where α have to be replaced by $\alpha\ F$, since the recursive calls will be applied to a term of type $\alpha\ F\ T$. The explicit universal quantification over α indicates that the blueprint needs to be truly polymorphic in α.

Bird and Paterson [9] observe that the above combinator is not practical, since the primitive iteration scheme it provides is very restrictive: it forces the type argument β of T to be fully polymorphic. In fact, neither the split function, nor a simple summation of a powerlist storing natural numbers can be expressed using that fold. To overcome the limitation, they propose a *generalized* fold of type

$$\forall X\ Y.\, (\forall \alpha.\, (\alpha\ X, \alpha\ F\ Y)\ G \to \alpha\ Y) \to (\forall \alpha.\, \alpha\ X\ F \to \alpha\ F\ X) \to \beta\ X\ T \to \beta\ Y$$

where the second argument enables recursive functions of a more refined type $\beta\ X\ T \to \beta\ Y$ by providing a distributive law $\mathsf{a} : \forall \alpha.\, \alpha\ X\ F \to \alpha\ F\ X$ which we call the (*argument*) *swapper*. Bird and Paterson require X and Y to be functors and the two function arguments b and a to fold to be natural transformations. The function $\mathsf{f} = \mathsf{fold}\ \mathsf{b}\ \mathsf{a}$ is then a natural transformation too and adheres to the characteristic equation

$$\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ (\mathsf{f} \circ \mathsf{map}_T\ \mathsf{a})\ g) \tag{2}$$

It is straightforward to allow functors X and Y to be of arbitrary arity n instead of 1. The function split can then be defined by setting, $n = 2$, $(\alpha, \beta)\ G = unit + \alpha \times \beta$, $\alpha\ F = \alpha \times \alpha$, $(\alpha, \beta)\ X = \alpha \times \beta$, $(\alpha, \beta)\ Y = \alpha\ plist \times \beta\ plist$, $\mathsf{a} = \mathsf{swap}$, and

$$\begin{aligned} \mathsf{b}\ (\mathsf{Inl}\ ()) &= (\mathsf{Nil}, \mathsf{Nil}) \\ \mathsf{b}\ (\mathsf{Inr}\ (ab, abs)) &= \mathsf{let}\ (as, bs) = abs \\ &\quad\ \mathsf{in}\ (\mathsf{Cons}\ (\mathsf{fst}\ ab)\ as, \mathsf{Cons}\ (\mathsf{snd}\ ab)\ bs) \end{aligned}$$

where Inl and Inr are the standard embeddings of $+$. For simplicity, the rest of the section assumes $n = 1$.

We propose an even more flexible fold combinator which replaces the functor F, which is fixed in the nonuniform datatype specification and shows up in the recursive calls, with another arbitrary functor V of the same arity as F (here, 1):

$$\forall X\ Y.\, (\forall \alpha.\, (\alpha\ X, \alpha\ V\ Y)\ G \to \alpha\ Y) \to (\forall \alpha.\, \alpha\ X\ F \to \alpha\ V\ X) \to \beta\ X\ T \to \beta\ Y$$

This allows the recursive calls to return a type $\alpha\ V\ Y$ instead of the fixed $\alpha\ F\ Y$. The combinator satisfies the same characteristic equation (2) (with the more general types).

All those expressive combinators for nonuniform types share one problem: in higher-order logic neither type constructor quantification nor type variable quantification that happens not at the outermost level is possible. Thus, it is impossible to define the fold constants for nonuniform datatypes in HOL.

Instead, we follow a similar route as for induction. We devise a *recursion procedure* that takes (here: unary) BNFs

V, X, Y,[2] a blueprint $\mathsf{b} : (\alpha\ X, \alpha\ V\ Y)\ G \to \alpha\ Y$ and a swapper $\mathsf{a} : \alpha\ X\ F \to \alpha\ V\ X$ as input and produces a function $\mathsf{f} : \alpha\ X\ T \to \alpha\ Y$ satisfying equation (2) as output.

Internally, the procedure defines a recursive function using b and a on the *raw* type and lifts it to the nonuniform type. To perform such a lifting for induction, the inductive property P had to be a polymorphic IAP term. For recursion, we require both b and a to be polymorphic injective-parametric terms, i.e., parametric only for relations that are graphs of injective function. This is a weaker assumption than Bird and Paterson's naturality assumption (e.g., $\mathsf{map}_F\ (\mathsf{map}_X\ f) = \mathsf{map}_X\ (\mathsf{map}_V\ f) \circ \mathsf{a}$ for a). On (bounded) natural functors injective-parametricity implies the weak naturality assumption that demands for the above equation to hold only for injective functions f. Consequently, f will also only be a natural transformation for injective functions. However, our construction is closed: If both b and a are fully parametric in some type parameters, f is fully parametric in those as well.

The definition of f proceeds in four steps. First, we define a shape type sh_V for V analogously to sh for F, including the constructors Leaf_V, Node_V, their inverses unLeaf_V, unNode_V, and the functions $\boxed{\mathsf{ok}}_V$, $\boxed{\uparrow}_V$, and $\boxed{\downarrow}_V$. Second, we lift a to shapes $\boxed{\mathsf{a}} : \Delta \to \alpha\ X\ sh \to \alpha\ sh_V\ X$ by recursion on the shadow:

$$\begin{aligned}\boxed{\mathsf{a}}\ [] &= \mathsf{map}_X\ \mathsf{Leaf}_V \circ \mathsf{unLeaf}\\ \boxed{\mathsf{a}}\ (1 \triangleleft u) &= \mathsf{map}_X\ \mathsf{Node}_V \circ \mathsf{a} \circ \mathsf{map}_F\ (\boxed{\mathsf{a}}\ u) \circ \mathsf{unNode}\end{aligned}$$

Third, we define a *raw* version of our function $\mathsf{f}_{raw} : \Delta \to \alpha\ X\ T \to \alpha\ sh_V\ Y$ by primitive recursion:

$$\mathsf{f}_{raw}\ u\ (\mathsf{Raw}\ g) = \mathsf{b}\ (\mathsf{map}_G\ (\boxed{\mathsf{a}}\ u)\ (\mathsf{map}_Y\ \mathsf{unNode}_V \circ \mathsf{f}_{raw}\ (1 \triangleleft u))\ g)$$

The generalization to sh_V in the return type of f_{raw} is similar to the generalization performed for induction. Finally, we define the function f as $\mathsf{f} = \mathsf{unLeaf}_V \circ \mathsf{f}_{raw}\ [] \circ \mathsf{Rep}$.

Figure 6 justifies why the above definitions make sense by proving equation (2). When reading the diagram, some of the arrows labeled by injective functions, such a $\mathsf{Leaf}_{(V)}$ and $\mathsf{Node}_{(V)}$ (possibly under further maps), need to be inverted for the diagram to make sense. Elements of the two highlighted types have shadow [1]. All other elements of types sh, sh_V, and raw occurring in the diagram have shadow [].

Equation (2) is the outermost pentagon, which is filled by commutative diagrams starting by unfolding the definitions of f ①, (twice), Ctor ②, and map_T ③ as well as the recursive specification of f_{raw} ④. The quadrilateral ⑤ follows from the naturality for injective functions (Leaf_V) of b and ⑥ from the recursive specification of $\boxed{\uparrow}_V$. The remaining commutative pentagon ⑦ crucially relates f_{raw} and $\uparrow$ (similar to Lemma 5 for map_{raw}). The proof of that fact follows by induction. Therefore, the property used in ⑦ for shadow [] needs to be generalized to an arbitrary u and requires an auxiliary fact about a and $\boxed{\uparrow}$ alongside with the facts that $\boxed{\mathsf{a}}$ and f_{raw} preserve $\boxed{\mathsf{ok}}\ u$ and $\mathsf{ok}\ u$. The proofs rely on the injective-parametricity of a and b.

Lemma 10:

1) $\boxed{\mathsf{ok}}\ u\ s \Longrightarrow \mathsf{pred}_{sh_V}\ (\boxed{\mathsf{ok}}_V\ u)\ (\boxed{\mathsf{a}}\ u\ s)$
2) $\mathsf{ok}\ u\ r \Longrightarrow \mathsf{pred}_{raw}\ (\boxed{\mathsf{ok}}_V\ u)\ (\mathsf{f}_{raw}\ u\ r)$
3) $\boxed{\mathsf{ok}}\ u\ s \Longrightarrow \boxed{\mathsf{a}}\ (u \triangleright 1)\ (\boxed{\uparrow}\ s) = \mathsf{map}_X\ \boxed{\uparrow}_V\ (\boxed{\mathsf{a}}\ u\ (\mathsf{map}_{sh}\ \mathsf{a}\ s))$
4) $\mathsf{ok}\ u\ r \Longrightarrow \mathsf{f}_{raw}\ (u \triangleright 1)\ (\uparrow s) = \mathsf{map}_Y\ \boxed{\uparrow}_V\ (\mathsf{f}_{raw}\ u\ (\mathsf{map}_{raw}\ \mathsf{a}\ s))$

Finally, we remark that asking for Y to be a natural functor, or more generally as in case of Bird and Paterson to be a (positive) functor is very restrictive, as it essentially disallows recursive functions with parameters. We generalize the whole construction to the case where $\alpha\ Y = \alpha\ Y_1 \to \alpha\ Y_2$ with Y_1 and Y_2 being natural functors. This allows first-order arguments or higher-order arguments that do not refer to α in their domain. This generalization is straightforward but technically involved.

B. Generalized Primitive Corecursion

We have carefully orchestrated the recursion procedure to work dually for codatatypes. Indeed, the corecursion procedure mainly reverses function arrows: It takes the injective-parametric blueprint $\mathsf{b} : \alpha\ Y \to \mathsf{b} : (\alpha\ X, \alpha\ V\ Y)\ G$ and swapper $\mathsf{a} : \alpha\ V\ X \to \alpha\ X\ F$ as inputs and produces the function $\mathsf{f} : \alpha\ Y \to \alpha\ X\ T$ as output, which satisfies

$$\mathsf{f}\ y = \mathsf{Cons}\ (\mathsf{map}_G\ \mathsf{id}\ (\mathsf{map}_T\ \mathsf{a} \circ \mathsf{f})\ (\mathsf{b}\ y)) \tag{3}$$

An example of the "orchestration" is the type and the definition of $\boxed{\mathsf{a}}$. For datatypes, the type is $\Delta \to \alpha\ X\ sh \to \alpha\ sh_V\ X$ and the definition is recursive on Δ. However, we could have defined it by recursion on the sh argument without the need for a Δ argument at all, i.e. $\boxed{\mathsf{a}} : \alpha\ X\ sh \to \alpha\ sh_V\ X$. For codatatypes, we would then fail to define the dual $\boxed{\mathsf{a}} : \alpha\ sh_V\ X \to \alpha\ X\ sh$, since there is no inductive argument on which we could recurse. The additional Δ argument restores the duality.

VII. Implementation

To add support for nonuniform types to Isabelle/HOL, we followed the same general strategy as previously [10]:

1) We formalized in Isabelle/HOL an abstract datatype example $\alpha\ T = \mathsf{Ctor}\ ((\alpha, \alpha\ F\ T)\ G)$ as well as a codatatype.
2) We developed ML functions that generalize the abstract examples to produce the derivations for a concrete set of mutual types with an arbitrary number of type variables and to derive the nonemptiness witnesses.
3) We developed ML functions that extend the results of step 2 to multiple curried constructors—the high-level view presented to users.
4) We developed the commands that process type and function definitions and that perform (co)induction.

For datatypes, step 1 starts by defining the type $\alpha\ T$, Ctor, and the BNF constants; then it derives theorems about them and registers T as a BNF. This registration relies on an Isabelle command that lifts the BNF structure of a type across an embedding–projection pair [6]. Induction is formalized by deriving a lemma $\mathsf{Q}\ t$ in terms of a fixed but unknown polymorphic predicate Q that is IAP and inductive (i.e., $(\forall x \in \mathsf{set}^2_G\ g.\ \mathsf{Q}\ x) \Longrightarrow \mathsf{Q}\ (\mathsf{Ctor}\ g)$ holds). Recursion is formalized as a function f defined such that the recursive

[2]Strictly speaking, the boundedness assumption are not needed for X and Y, which implies that α *set* is permitted to occur in those type expressions.

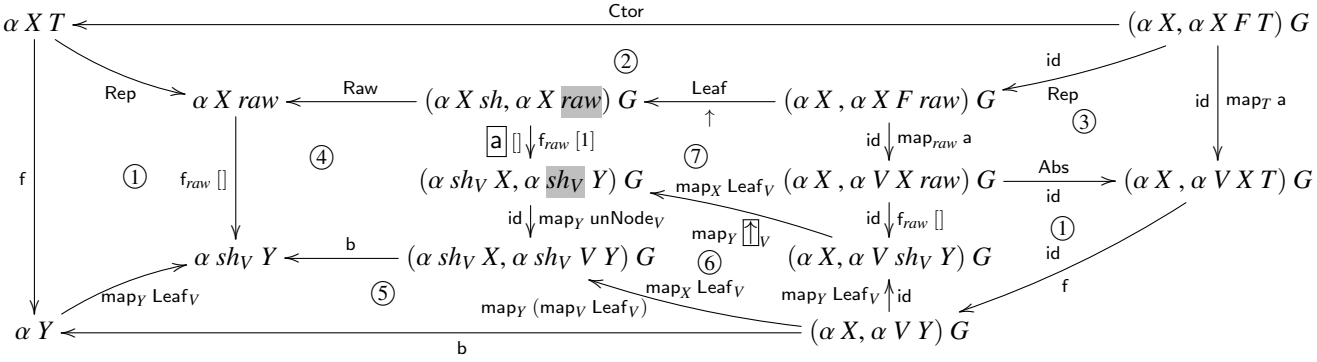

Fig. 6. Proof of the recursive specification of f

equation $\mathsf{f}\ (\mathsf{Ctor}\ g) = \mathsf{b}\ (\mathsf{map}_G\ \mathsf{id}\ (f \circ \mathsf{map}_T\ \mathsf{a})\ g)$ holds for a fixed injective-parametric blueprint b and swapper a.

The code for step 2 constructs the low-level types, terms, and lemma statements presented in Sections III to VI and proves the lemmas using specialized *tactics*—ML programs that generalize the proofs from the formalization. In principle, the tactics should always succeed, but it is necessary to execute them to obtain the highest level of trustworthiness. Assuming Isabelle's inference kernel is correct, bugs in the new commands might lead to run-time failures but never to logical inconsistencies. For step 3, we were able to generalize and reuse the infrastructure for uniform types that performs the same lifting from low to high level [10, Sections 3–6].

Step 4 takes the form of six main commands available to the users and making definitions and reasoning about nonuniform types nearly as convenient as for uniform types.

The `nonuniform_(co)datatype` commands can be used to define nonuniform types. For example, the following definition introduces a type of λ-terms over variables drawn from α, with De Bruijn notation for bound variables [8]:

```
nonuniform_datatype α tm =
  Var α | App (α tm) (α tm) | Lam ((unit + α) tm)
```

Entering a λ-abstraction (Lam) creates a new variable, which is accommodated by the extended type $unit + \alpha$ consisting of the values Inl () (the new variable) and Inr x for all $x : \alpha$. The command performs the type construction and computes a nonemptiness witness. Then it defines the constructors Var, App, Lam and corresponding destructors and derives characteristic theorems about the constructors, the destructors, and the BNF constants map_{tm}, pred_{tm}, rel_{tm}, and set_{tm}.

The `nonuniform_prim(co)recursive` commands allow users to define primitively (co)recursive functions, by specifying their (co)recursive equations.[3] For example, the following definition introduces a function join that "flattens" a term whose variables are themselves terms:

```
nonuniform_primrecursive join : α tm tm → α tm where
  join (Var x) = x
| join (App s t) = App (join s) (join t)
| join (Lam u) = Lam (join (map_tm (λx. case x of
    Inl () ⇒ Var (Inl ()) | Inr y ⇒ map_tm Inr y) u))
```

The command extracts blueprints and swappers from the user-specified equations and emits parametricity proof obligations that must be discharged by the user. In the example, the swapper is the λ-expression of type $unit + \alpha\ tm \to (unit + \alpha)\ tm$ that is passed to the outer map_{tm}. Once the proofs are complete, the command derives a low-level characteristic theorem about the defined function. Then it derives the equations specified by the user from this theorem.

One of the most basic operations on λ-terms is substitution: $\mathsf{subst} : (\alpha \to \beta\ tm) \to \alpha\ tm \to \beta\ tm$. Due to the limitation that arguments to recursive functions must be BNFs, we cannot define higher-order functions like subst that depend on a type variable that changes in the recursive calls. But we can define subst as a composition: $\mathsf{subst}\ \sigma = \mathsf{join} \circ \mathsf{map}_{tm}\ \sigma$.

The `nonuniform_(co)induct` commands can be used to prove a lemma (or a set of lemmas for mutual definitions) by (co)induction. For example, the command

```
nonuniform_induct s in subst_subst:
  subst τ (subst σ s) = subst (subst τ ∘ σ) s
```

emits proof obligations for parametricity and the three cases of the induction on s. Often, the pararametricity proofs can be delegated to Isabelle's Transfer tool [26]. Once the obligations are discharged, the stated property is derived and stored under the specified name (*subst_subst*). For the technical reason explained in Section V, the derivation can be performed only by an Isabelle command, not by a proof method as is done for uniform (co)datatypes [10]. The main advantage of proof methods is that they can be invoked on an arbitrary proof goal in the middle of a proof.

We conclude with a codatatype example: We prove two alternative definitions of the constant powerstream equivalent. All required proofs are fully automatic after specifying the trivial bisimulation relation $\mathsf{R}\ l\ r \Leftrightarrow \exists x\ xs.\ l = \mathsf{const}\ x \wedge r = \mathsf{map}_{pstream}\ (\lambda\_.\ x)\ xs$ in the coinduction proof.

```
nonuniform_codatatype α pstream =
  Cons α ((α × α) pstream)

nonuniform_primcorecursive const : α → α pstream
  const x = Cons x (const (x, x))
```

[3]The implementation of these two commands is incomplete at the time of this writing. We do not foresee any difficulties beyond those which we met for the other commands and expect to finish the implementation in the weeks following the submission deadline. Our archive [11] will be updated.

nonuniform_coinduct R in *const_alt*:
  const $x = \mathsf{map}_{pstream}\ (\lambda\_ : \alpha.\ x)\ xs$

VIII. Discussion and Related Work

a) Inspiration: For representation, we generalized Okasaki's construction [37] to arbitrary datatypes. Nordhoff et al. [36] have used this construction partially (defining the *sh* and *raw* types but without introducing a new nonunform type) in their Isabelle/HOL formalization of 2-3 finger trees. The corresponding reduction of nonuniform to uniform codatatypes appears not to have been studied in the literature.

For recursion, we refined Bird and Paterson's generalized fold combinators [9] in several ways, including weakening the parametricity/naturality condition and enabling non-functor target domains. In turn, Bird and Paterson had improved on the standard sheaf-functor approach from category theory [29].

Our (co)induction principles take advantage of the BNF structure including set operators and relators, and form a lighter alternative to fibration-based approaches [17], [22] for the category of sets and functions.

b) Comparison with Other Proof Assistants: Our work shows that nonuniform (co)datatypes and the associated polymorphic (co)recursion [21], [34] can be supported in the minimalistic rank-one polymorphic framework of HOL, and therefore made available in HOL-based provers, which cover about half of the theorem proving community.

The dependent type theory (DTT) camp, represented by theorem provers such as Agda, Coq, Matita, and Lean, has sophisticated type systems built into their mechanized logic, including native nonuniform datatypes. Several case studies in these provers exploit nonuniformity [4], [15], [25], [35], [42].

Compared with the DTT systems, our support for nonuniform types in HOL has some limitations. First, dependent families of (nonuniform) types cannot be expressed in HOL. Second, Agda supports self-nested (co)datatypes. For example, a definition such as $\alpha\ bush = \mathsf{BNil} \mid \mathsf{BCons}\ \alpha\ (\alpha\ bush\ bush)$ is beyond scope of our results, since the recursive occurrence of *bush* is nested in itself. Third, our (co)induction principles have some restrictions concerning (a weak form of) parametricity. The reason is that we cannot perform well-founded induction across types in HOL. While practical predicates about functional programs obey them, the restrictions are not necessary sematically. Appendix D presents an axiomatic extension that allows HOL to "see" cross-type. However, adding axioms, no matter how provably correct they may be, goes against the main tenets of HOL.

Our approach has some advantages as well, stemming from its category-theory-awareness. First, arbitrary parameter types, not just (co)datatypes, can be plugged in the specifications for nonuniform (co)datatypes, either inside or outside of the recursive occurrences in the specification. For example, the type *stree* from Section I is possible because the type $\alpha\ fset$, of finite sets is a BNF. This is not possible with DTT, where datatypes are restricted to a predefined grammar.

Second, since the foundational approach compels us to maintain the functorial structure to justify fixpoint definitions, users can enjoy default map functions and relators, as well as some polytypic properties either out of the box or within the immediate reach. Thus, our nonuniform recursion principle delivers parametric functions, i.e., natural transformations. Moreover, the fusion laws [9] (Appendix C), known to be important in reasoning about functional programs, rely heavily on functoriality and naturality, and they are immediate in our framework. In contrast, in DTT, very little structure is available for nonuniform datatypes after definition. In particular, map functions and relators are missing and can be difficult to add.

c) Other Work: The pioneering work of Bird and his collaborators on nonuniform datatypes [7], [8], [9] has been extended into several directions, including structures for efficient functional programming [23], [24], [30], datatypes with references [16], as well as work directly relevant for DTT proof assistants: reduction to W-types and container types [1], typed term rewriting frameworks for total programming [2], [3], [31], intensional-DTT induction [32]. Our current contribution was concerned with bootstrapping nonuniform datatypes in HOL on a sound and compositional basis. Only time will tell if Isabelle/HOL users, or more generally the HOL community of users and researchers, will embrace nonuniform datatypes and their applications to a similar scale as in advanced programming languages and type theories.

Acknowledgment: We thank Peter Lammich for pointing us to his work on finger trees and for formalizing Okasaki's construction for powerlists and Johannes Hölzl for explaining us Lean's support for nonuniform datatypes.

References

[1] M. G. Abbott, T. Altenkirch, and N. Ghani. Representing nested inductive types using W-types. In *ICALP 2004*, vol. 3142 of *LNCS*, pp. 59–71. Springer, 2004.

[2] A. Abel and R. Matthes. Fixed points of type constructors and primitive recursion. In *CSL 2004*, vol. 3210 of *LNCS*, pp. 190–204. Springer, 2004.

[3] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005.

[4] N. Benton, C. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.

[5] S. Berghofer and M. Wenzel. Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In *TPHOLs '99*, vol. 1690 of *LNCS*, pp. 19–36, 1999.

[6] J. Biendarra. *Functor-Preserving Type Definitions in Isabelle/HOL*. B.Sc. thesis, Technische Universität München, 2015.

[7] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *MPC'98*, vol. 1422 of *LNCS*, pp. 52–67. Springer, 1998.

[8] R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91, 1999.

[9] R. S. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Asp. Comput.*, 11(2):200–222, 1999.

[10] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.

[11] J. C. Blanchette, F. Meier, A. Popescu, and D. Traytel. Formalization and implementation accompanying this paper. https://people.mpi-inf.mpg.de/~jblanche/nonuniform.tar.gz, 2016.

[12] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In *ICFP 2015*, pp. 192–204. ACM, 2015.

[13] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In *ESOP 2015*, vol. 9032 of *LNCS*, pp. 359–382. Springer, 2015.

[14] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5(2):56–68, 1940.

[15] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL 2008*, pp. 133–144. ACM, 2008.

[16] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *TFP 2006*, vol. 7 of *Trends in Functional Programming*, pp. 173–188. Intellect, 2006.

[17] N. Ghani, P. Johann, and C. Fumex. Generic fibrational induction. *Logical Methods in Computer Science*, 8(2), 2012.

[18] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[19] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In *HUG '93*, vol. 780 of *LNCS*, pp. 141–154. Springer, 1994.

[20] J. Harrison. Inductive definitions: Automation and application. In *TPHOLs '95*, vol. 971 of *LNCS*, pp. 200–213. Springer, 1995.

[21] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[22] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[23] R. Hinze. Efficient generalized folds. In *Workshop on Generic Programming*, pp. 1–16, 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

[24] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[25] A. Hirschowitz and M. Maggesi. Nested abstract syntax in Coq. *J. Autom. Reasoning*, 49(3):409–426, 2012.

[26] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In *CPP 2013*, vol. 8307 of *LNCS*, pp. 131–146. Springer, 2013.

[27] O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In *ITP 2015*, vol. 9236 of *LNCS*, pp. 234–252. Springer, 2015.

[28] O. Kunčar and A. Popescu. Comprehending Isabelle/HOL's consistency. In *ESOP*, 2017. To appear. Preprint available at http://andreipopescu.uk/pdf/compr_IsabelleHOL_cons.pdf.

[29] J. Lambek. Subequalizers. *Canadian Mathematical Bulletin*, 13(1):337–349, 1970.

[30] C. E. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Asp. Comput.*, 16(1):19–35, 2004.

[31] R. Matthes. Recursion on nested datatypes in dependent type theory. In *CiE 2008*, vol. 5028 of *LNCS*, pp. 431–446. Springer, 2008.

[32] R. Matthes. An induction principle for nested datatypes in intensional type theory. *J. Funct. Program.*, 19(3-4):439–468, 2009.

[33] T. F. Melham. Automating recursive type definitions in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 341–386. Springer, 1989.

[34] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Symposium on Programming*, vol. 167 of *LNCS*, pp. 217–228. Springer, 1984.

[35] G. Naves and A. Spiwack. Balancing lists: A proof pearl. In *ITP 2014*, vol. 8558 of *LNCS*, pp. 437–449. Springer, 2014.

[36] B. Nordhoff, S. Körner, and P. Lammich. Finger trees. In *Archive of Formal Proofs*. http://afp.sf.net/entries/Finger-Trees.shtml, 2010.

[37] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[38] L. C. Paulson. A formulation of the simple theory of types (for Isabelle). In *COLOG-88*, vol. 417 of *LNCS*, pp. 246–274. Springer, 1990.

[39] A. Pitts. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, chapter The HOL Logic, pp. 191–232. In Gordon and Melham [18], 1993.

[40] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP '83*, pp. 513–523, 1983.

[41] J. J. M. M. Rutten. Relators and metric bisimulations. *Electr. Notes Theor. Comput. Sci.*, 11:252–258, 1998.

[42] M. Sozeau. PROGRAM-ing finger trees in Coq. In *ICFP'07*, pp. 13–24. ACM, 2007.

[43] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *LICS 2012*, pp. 596–605. IEEE Computer Society, 2012.

[44] P. Wadler. Theorems for free! In *FPCA '89*, pp. 347–359. ACM, 1989.

APPENDIX

A. (Weak) Parametricity and Naturality

$\mathsf{c} : \overline{\alpha}\ F$ parametric $\mathsf{rel}_F\ \overline{R}\ \mathsf{c}\ \mathsf{c}$ for all $\overline{R}$

$\mathsf{f} : \overline{\alpha}\ F \to \overline{\alpha}\ G$ parametric (P) $(\mathsf{rel}_F\ \overline{R} \Rightarrow \mathsf{rel}_G\ \overline{R})\ \mathsf{f}\ \mathsf{f}$ for all $\overline{R}$

Function space relator $R \Rightarrow S$ defined as $\lambda f\ g.\ (\forall a\ b.\ R\ a\ b = S\ (f\ a)\ (g\ b))$. In other words two functions are related by $R \Rightarrow S$ if for all R-related inputs, the outputs are S-related.

$\mathsf{f} : \overline{\alpha}\ F \to \overline{\alpha}\ G$ injective-parametric (IP) $(\mathsf{rel}_F\ \overline{R} \Rightarrow \mathsf{rel}_G\ \overline{R})\ \mathsf{f}\ \mathsf{f}$ for all $\overline{R}$ that are graphs of injective functions, i.e., left-total, singlevalued relations.

f natural transformation (NAT) $\mathsf{f} \circ \mathsf{map}_F\ \overline{g} = \mathsf{map}_G\ \overline{g} \circ \mathsf{f}$ for all $\overline{g}$

predicate $\mathsf{P} : \overline{\alpha}\ F \to \mathit{bool}$ (injective-)parametric $(\mathsf{rel}_F\ \overline{R} \Rightarrow (=))\ \mathsf{P}\ \mathsf{P}$ for all $\overline{R}$ (that are graphs of injective functions).

predicate $\mathsf{P} : \overline{\alpha}\ F \to \mathit{bool}$ injective-monotone parametric (IMP) $(\mathsf{rel}_F\ \overline{R} \Rightarrow (\Longrightarrow))\ \mathsf{P}\ \mathsf{P}$ for all $\overline{R}$ that are graphs of injective functions

predicate $\mathsf{P} : \overline{\alpha}\ F \to \mathit{bool}$ injective-antitone parametric (IAP) when $(\Longrightarrow)$ is replaced by $(\Longleftarrow)$

P = NAT

P $\Longrightarrow$ IP

IP $\Longrightarrow$ IMP

IP $\Longrightarrow$ IAP

IMP and IAP $\Longrightarrow$ IP

B. Proof Sketch of Theorem 6

Since the types T_i do not exist yet, we need to work on the representation types raw_i, and prove a property about the predicates ok_i that represent the to-be-defined types T_i. For this, we introduce a refinement of the notion of witness: Given a shadow $u : \Delta$, a set $I \subseteq [\mathbf{i}]$ is an *u-witness* for raw_i if, for all sets $\overline{A}$, $\forall k \in I.\ A_k \neq \varnothing$ implies $\exists r \in \mathit{raw}_i\ \overline{A}.\ \mathsf{ok}_i\ u\ r$. We prove:

(3) $\mathsf{Lang}_i(\mathsf{Gr})$ (or $\mathsf{Lang}_{\infty,i}(\mathsf{Gr})$) is a perfect sets of $[]$-witnesses for the (co)datatype raw_i.

To prove (3), we are looking for a connection between the grammar Gr and raw_i's (co)recursive specification, in the direction of the destructor

$$\overline{\alpha}\ \mathit{raw}_i \xrightarrow{\mathsf{unRaw}_i} (\overline{\overline{\alpha}\ \mathit{sh}}, \overline{\alpha}\ \mathit{raw}_{\sigma(1)}, \cdots, \overline{\alpha}\ \mathit{raw}_{\sigma(\mathbf{j})})\ G_i$$

By the definition of ok_i, if $r : \overline{\alpha}\ \mathit{raw}_i$ is such that $\mathsf{ok}_i\ u\ r$, then $\mathsf{dtor}_i\ r$ has, for each $j \in [\mathbf{j}]$, its j-components $r' : \overline{\alpha}\ \mathit{raw}_{\sigma(j)}$ satisfying $\mathsf{ok}_i\ (j \triangleleft u)\ r'$. As discussed in Section III-B, the shadow increment between u and $j \triangleleft u$ encodes the application of the components $\overline{F_j} = (F_{j1}, \ldots, F_{j\mathbf{k}})$, reflected in the grammar's productions of type 2. This suggests a recursive translation of shadows into polywits:

- $\gamma_k\ [] = \{\{k\}\}$
- $\gamma_k\ (j \triangleleft u) = (\gamma_1\ u, \ldots, \gamma_{\mathbf{k}}\ u) \cdot \mathscr{J}(F_{jk})$

Now we can formulate a generalization of (3), taking into account arbitrary shadows, not just $[]$. For each nonterminal x, we write $\mathsf{Lang}_x(\mathsf{Gr})$ (or $\mathsf{Lang}_{\infty,x}(\mathsf{Gr})$) for the language (co)generated by x.

(4) For all $u : \Delta$, $\mathsf{Lang}_{(\gamma_k\ u, \ldots, \gamma_k\ u)\ t_i}(\mathsf{Gr})$ (or $\mathsf{Lang}_{\infty,(\gamma_k\ u, \ldots, \gamma_k\ u)\ t_i}(\mathsf{Gr})$) is a perfect set of u-witnesses for the (co)datatype raw_i.

Finally, (4) can be proved using standard (uniform) (co)induction and (co)recursion, moving back and forth between Gr-derivation trees and the raw_i's.

For datatypes.: That every $I \in \mathsf{Lang}_{(\gamma_k\, u,\ldots,\gamma_k\, u)\, t_i}(\mathsf{Gr})$ is an u-witnesses follows by structural induction on its derivation tree in Gr. Conversely, that for every u-witness J, we have $I \subseteq J$ for some $I \in \mathsf{Lang}_{(\gamma_k\, u,\ldots,\gamma_k\, u)\, t_i}(\mathsf{Gr})$ follows by induction on the definition of ok_i.

For codatatypes.: To prove that every $I_0 \in \mathsf{Lang}_{\infty,(\gamma_k\, u_0,\ldots,\gamma_k\, u_0)\, t_{i_0}}(\mathsf{Gr})$ is an u_0-witness, we let $\overline{A}$ be such that $\forall i \in I_0.\, A_i \neq \varnothing$. With u_0, i_0, I_0 and $\overline{A}$ fixed, let Tr be a (possibly infinite) derivation tree of $(\gamma_k\, u_0, \ldots, \gamma_k\, u_0)\, t_{i_0}$ in Gr—thus having I_0 as the set of terminals on its frontier. Let $\Delta_{u_0,i}$ consists of all shadows having u_0 as a prefix and such that the nonterminal $(\overline{\gamma}\, u)\, t_i$ occurs in the tree Tr, where we write $\overline{\gamma}\, u$ for $(\gamma_1\, u, \ldots, \gamma_{\mathbf{k}}\, u)$.

Mutually corecursively (by primitive *raw*-corecursion), we define the functions $w_i : \Delta_{u_0,i} \to \overline{\alpha}\ raw_i$, by

$$w_i\, u = \mathsf{map}_{G_i}\ \overline{\mathsf{id}}\ (w_{\sigma(1)}, \ldots, w_{\sigma(\mathbf{j})})\ g_u$$

where the element $g_u \in (\overline{\overline{\alpha}\ sh}, \Delta_{u_0,\sigma(1)}, \cdots, \Delta_{u_0,\sigma(\mathbf{j})})\ G_i$ is defined as follows: Let $(\overline{\gamma}\, u)\, t_i \Longrightarrow \Gamma_I$ be the (type 2) production in Tr corresponding to the terminal $(\overline{\gamma}\, u)\, t_i$.

- From the definition of Γ_J it follows that, for each j such that $\mathbf{k} + j \in J$, the nonterminal $((\overline{\gamma}\, u) \cdot \mathscr{I}(F_{j1}), \ldots, (\overline{\gamma}\, u) \cdot \mathscr{I}(F_{j\mathbf{k}}))\, t_{\sigma(j)}$, i.e., $(\gamma_1(j \triangleleft u), \ldots, \gamma_{\mathbf{k}}(j \triangleleft u))\, t_{\sigma(j)}$, is also in Tr; hence $j \triangleleft u \in \Delta_{u_0,\sigma(j)}$.
- Also from the definition of Γ_J it follows that, for each $k \in [\mathbf{k}] \cap J$, the nonterminal $\gamma_k\, u$ is also in Tr. Since only type 1 productions are applicable to polywit nonterminals, let $\gamma_k\, u \to I$ be the production from Tr applied to $\gamma_k\, u$. Then I is included in Tr's frontier, i.e., $I \subseteq I_0$. Then, picking some elements $a_i \in A_i$ for $i \in I$, we can define shapes $s_k \in sh_k\ \overline{A}$ for each $k \in [\mathbf{k}]$ that are full trees, i.e., that $\boxed{\mathsf{ok}}_u\ s_i$ holds.

Thus, we have constructed the elements $j \triangleleft u \in \Delta_{u_0,\sigma(j)}$ for each j such that $\mathbf{k} + j \in J$ and $s_i \in sh_k\ \overline{A}$ (in particular, $s_i : \overline{\alpha}\ sh_k$) for each $k \in [\mathbf{k}]$. Since J is a witness for G_i, we obtain our desired element $g_u \in (\overline{\overline{\alpha}\ sh}, \Delta_{u_0,\sigma(1)}, \cdots, \Delta_{u_0,\sigma(\mathbf{j})})\ G_i$, which concludes the definition of the w_i's. Because of our choices in the definition, it is now routine to prove:

- by rule coinduction on the definition of the ok_i's, that $\mathsf{ok}_i\ u\ (w_i\ u)$ holds;
- by rule induction on the definition of the set operators for *raw*, that $\mathsf{set}_{raw,k}(w_i\, u) \subseteq A_i$ holds, which means that $w_i\, u \in raw_i\ \overline{A}$ holds.

This concludes the proof that I_0 is a witness.

Conversely, to prove that for every u-witness J, we have $I \subseteq J$ for some $I \in \mathsf{Lang}_{\infty,(\gamma_k\, u,\ldots,\gamma_k\, u)\, t_i}(\mathsf{Gr})$, we construct a (possibly infinite) derivation tree whose frontier includes J. The construction proceeds corecursively, by extracting each time the next production to be applied from J and the G_i's witnesses. □

C. Fusion Laws

We fix a nonuniform datatype $\alpha\ T = (\alpha, \alpha\ F\ T)\ G$. Let us write $\mathsf{NURec}(X, Y, \mathsf{a}, \mathsf{b})$ for the polymorphic function $\mathsf{f} : \alpha\ X\ T \to \alpha\ Y$ defined by nonuniform recursion from a blueprint $\mathsf{b} : (\alpha\ X, \alpha\ V\ Y)\ G \to \alpha\ Y$ and a swapper $\mathsf{a} : \alpha\ X\ F \to \alpha\ V\ X$, as in Section VI-A. (Of course, NURec cannot be a HOL combinator—it is just a meta-level notation.)

Theorem 11: The following hold:

Fold Fusion: If $\kappa : \alpha\ Y \to \alpha\ Y'$ is such that $\kappa \circ \mathsf{b} = \mathsf{b}' \circ \mathsf{map}_G\ \mathsf{id}\ \kappa$, then $\kappa \circ \mathsf{NURec}(X, Y, \mathsf{a}, \mathsf{b}) = \mathsf{NURec}(X, Y', \mathsf{a}, \mathsf{b}')$.

Map Fusion: If $\kappa : \alpha\ X' \to \alpha\ X$ is such that $\kappa \circ \mathsf{a}' = a \circ \kappa$, then $\mathsf{NURec}(X, Y, \mathsf{a}, \mathsf{b}) \circ \mathsf{map}_T\ \kappa = \mathsf{NURec}(X', Y, \mathsf{a}', \mathsf{b})$.

Proof. TODO: By nonuniform induction; things are suitably parametric (equality of parametric / IAP functions); maybe draw diagrams, writing f and f′ for the recursively defined functions. □

The duals of the funsion laws hold when $\alpha\ T \overset{\infty}{=} (\alpha, \alpha\ F\ T)\ G$ is a uniform codatatype as in Section VI-B.

D. Cross-Type Induction Schema

As discussed in the paper, a main restriction of our work is induction for nonuniform types, where we require IA-parametricity of the predicate. Here, we show how a gentle, provably consistent axiomatic extension of HOL removes this restriction. The axiom does not refer to nonuniform datatypes, or the intricate construction leading to them, or even to BNFs. Rather, it is a general-purpose axiom for cross-type well-founded induction and recursion.

We fix the types $\alpha\ T$, $\alpha\ F$ and M (with the notations T and F not connected to nonuniform datatypes).

Let $\mathsf{P} : \alpha\ T \to \mathit{bool}$ be a polymorphic predicate, for which we want to prove $\forall \alpha.\, \forall t : \alpha\ T.\, \mathsf{P}\ t$ A natural approach would be induction using a measure $\mathsf{m} : \alpha\ T \to M$ which decreases w.r.t. a well-founded relation $\mathsf{r} : M\ \mathit{set} \to M\ \mathit{set} \to \mathit{bool}$. But what if the measure decreases by changing the type, as in $\mathsf{r}\ (\mathsf{m}\ t')\ (\mathsf{m}\ t)$, where $t : \alpha\ T$ and $t' : \alpha\ F\ T$? This is still acceptable, since well-foundedness should still operate across the types $\alpha\ F^n\ T$. Formally, we would like to have the following rule, where wf r states that r is well-founded.

$$\frac{\begin{array}{l}\forall \alpha.\, \forall t : \alpha\ T.\\ \quad \mathsf{wf}\ \mathsf{r} \wedge (\forall t' : \alpha\ F\ T.\, \mathsf{r}\ (\mathsf{m}\ t')\ (\mathsf{m}\ t) \Longrightarrow \mathsf{P}\ t')\\ \quad \Longrightarrow \mathsf{P}\ t\end{array}}{\forall \alpha.\, \forall t : \alpha\ T.\, \mathsf{P}\ t}\ \mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$$

In HOL, this type of induction in HOL across varying types is impossible to justify. The problem is the change in type during the descent: the elements t' smaller than t (via m, w.r.t. r) do not dwell the same type as t, $\alpha\ T$, but a different type $(\overline{\alpha}\ F)\ T$, And induction in HOL across varying types is impossible (unless, as we have seen, we require parametricity). However, it is easy to see that the rule is safe:

Theorem 12: The rule schema WFInd is sound in the standard models of HOL [39] and in the ground models

of Isabelle/HOL [27], hence is consistent with HOL and Isabelle/HOL.[4]

Proof. A standard model of HOL fixes a universe $\mathscr{U}$ of sets with good slosure properties (e.g., closed under function spaces) and interprets a type constructor such as T as a function on this universe, $[T] : \mathscr{U} \to \mathscr{U}$, a type such as M as an element of the universe, $[M] \in \mathscr{U}$, etc. Moreover, a polymorphic constant such as $\mathsf{m} : \alpha\ T \to M$ is intepreted as a $\mathscr{U}$-indexed family $([\mathsf{m}]_A)_{A \in \mathscr{U}}$. Crucially, it interprets the function-space type constructor as the set of all functions between the interpretation of its arguments and the type *nat* os a countable set $[nat]$, which with $[0]$ and $[\mathsf{Suc}]$ is isomorphic to the natural numbers. This means that the scheme WFInd can be justified inside $\mathscr{U}$ as follows: Assuming its conclusion is false and repeatedly using its hypothesis, we obtain the infinite seqeuences $(A_i)_i$ and $(b_i)_i$ such that $A_{i+1} = [F](A_i)$, $b_i \in [T](A_i)$, $[\mathsf{m}](A_i)(b_i) = [\mathsf{True}]$ and $[\mathsf{r}]\,([\mathsf{m}](A_{i+1})(b_{i+1}))\,([\mathsf{m}](A_i)(b_i)) = [\mathsf{True}]$. Taking $c_i = [\mathsf{m}](A_i)(b_i)$, this gives us an infinite sequence $(c_i)_i$ such that $c_i \in [M]$ and $[\mathsf{r}]\,(c_{i+1})\,(c_i) = [\mathsf{True}]$. Thanks to standardness, $(c_i)_i$ yields a witness for the formula $\exists c' : nat \to M.\ \forall i : nat.\ \mathsf{r}\,(c'\ i)\,(c'\ i)$, which therefore holds in the model. This is in contradiction with the fact that $[\mathsf{wf}\ \mathsf{r}]$ also holds in the model.

A ground model of Isabelle/HOL only interprets the ground (monomorphic) types and terms, again with a standard interpretation for functions and numbers. A formula is true in such a model iff all its ground substitutions are true. For example, $\forall x : \alpha.\ x = x$ is deemed true because, for all ground types K, the ground formula $\forall x : K.\ x = x$ is true. The argument for why the schema WFInd is sound is similar to the case of standard HOL models, but employing ground types K_i instead of the sets A_i. □

With this addition, we can remove the parametricity requirement from Theorem 8:

Theorem 13: The Ind schema is derivable in HOL enriched with the WFInd schema.

Proof. The derivation takes place by instantiating the parameters of $\mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$ using those of $\mathsf{Ind}^{\mathsf{P}}_T$. We take:

- T and F to be the nonuniform datatype and its nesting BNF
- M to be the datatype $M = \mathsf{MCons}\,(unit, M)\,G$
- r to be the immediate subterm relation associated to M, namely $\mathsf{r} = \{(m', m) \mid m' \in \mathsf{set}_{G2}\,(\mathsf{MCons}\,m)\}$
- m to be the composition $\mathsf{rawmeas} \circ \mathsf{Rep}_T$, where $\mathsf{Rep}_T : \alpha\ T \to \alpha\ raw$ is the representation function for T and $\mathsf{rawmeas} : \alpha\ raw \to M$ sends any r to its recursive depth:

$$\mathsf{rawmeas}\,(\mathsf{Raw}\ g) = \mathsf{MCons}\,(\mathsf{map}_G\ (\lambda\_.())\ \mathsf{rawmeas}\ g)$$

It is not hard to verify the assumptions of $\mathsf{WFInd}_{T,F,\mathsf{r},\mathsf{m},\mathsf{P}}$. □

In summary, the unrestricted versions of nonuniform induction is made available via a consistent axiomatic extension of HOL. The users can choose between enabling this axiom or using the more restricted principle we were able to prove entirely in HOL.

[4]The reason why we treat Isabelle/HOL specially is that it allows ad hoc overloading of constants intertwined with type definitions, which is problematic in the standard HOL semantics [27], [28].